MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

The Multiplexed Information and Computing Service: Programmers' Manual

PART I

INTRODUCTION TO MULTICS

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006

Revision: 14

Date: 9/30/73

P R E F A C E

The Multics project was begun in 1964 by the Computer Systems Research group of M.I.T. Project MAC. The goal was to create a prototype of a computer utility. In 1965, the project became a cooperative venture of M.I.T. Project MAC, the General Electric Company Computer Department (now Honeywell Information Systems Inc.) and the Bell Telephone Laboratories. In 1969, at the end of the research phase of the project, Bell Telephone Laboratories ended its active involvement. Also in 1969, the M.I.T. Information Processing Center began to offer Multics as a computing service within the M.I.T. community. In 1973, after developing a new hardware base for Multics, Honeywell announced that it would market Multics as a commercial product.

The Multics system owes its genesis to a small team of computer scientists who had the vision to lay out a plan which for 1965 was startlingly ambitious. This team consisted of the authors of a set of landmark papers published in the 1965 Fall Joint Computer Conference. Since that time literally hundreds of individuals have contributed to the Multics project, but no individual stands out so clearly in contribution as does Professor Fernando J. Corbato, who took responsibility for guiding the design and implementation of Multics from its initial proposal through to the time when Honeywell began to market the system.

The project would not have been possible without the considerable commitments of resources and talent made by the several organizations. These commitments were made on the recommendations of Professor Robert M. Fano, then director of Project MAC, Dr. John W. Weil, then of General Electric, and Dr. Edward E. David, Jr., then of the Bell Telephone Laboratories. The Information Processing Techniques office of the Advanced Research Projects Agency provided the primary financial support to Project MAC, and the Office of Naval Research provided contract supervision.

This technical report is a snapshot of The Introduction to the users' manual for the Multics system. It is being published in this form as a convenient method of communication with researchers and students of computer system design. The complete current users' manual is available in three updateable volumes from the M.I.T. Information Processing Center, or in a five-volume package from Honeywell. The present report represents Volume I of the three-volume version. The construction of the users' manual was also a team effort, with dozens of contributors. This manual has had the good fortune to have been maintained by a succession of three excellent editors, Michael A. Padlipsky, Laurie J. Haron, and Karolyn J. Martin, each of whom put in endless hours developing a general consistency of style, format, and presentation, so as to make the usefulness of the manual evenly predictable.

This preface can acknowledge only a few particular contributions. More detailed acknowledgements for specific contributions will be found among the 29 technical papers that have been published about Multics, some of which are reproduced in chapter two of this report. Unfortunately, in a team effort, complete and accurate acknowledgement is impossible, except by thanking all the members of the team for their intense devotion to the business of getting Multics designed and implemented.

Jerome H. Saltzer, Head
Computer Systems Research Division
M.I.T. Project MAC
September 21, 1973

F O R E W O R D

PLAN OF THE MULTICS PROGRAMMERS' MANUAL

September 30, 1973

The Multics Programmers' Manual (MPM) is the primary reference manual for user and subsystem programming on the Multics system. It is divided into three major parts:

Part I: Introduction to Multics

Part II: Reference Guide to Multics

Part III: Subsystem Writers' Guide to Multics

Part I is an introduction to the properties, concepts, and usage of the Multics system. Its four chapters are designed for reading continuity rather than for reference or completeness. Chapter 1 provides a broad overview. Chapter 2 goes into the concepts underlying Multics. Chapter 3 is a tutorial guide to the mechanics of using the system, with illustrative examples of terminal sessions. Chapter 4 provides a series of examples of programming in the Multics environment.

Part II is a self-contained comprehensive reference guide to the use of the Multics system for most users. In contrast to Part I, the Reference Guide is intended to document every detail and to permit rapid location of desired information, rather than to facilitate cover-to-cover reading.

Part II is organized into ten sections, of which the first eight systematically document the overall mechanics, conventions, and usage of the system. The last two sections of the Reference Guide are alphabetically organized lists of standard Multics commands and subroutines, respectively, giving details of the calling sequence and the usage of each.

Several cross-reference facilities help locate information in the Reference Guide:

- The table of contents, at the front of the manual, provides the name of each section and subsection and an alphabetically ordered list of command and subroutine names.
- A comprehensive index (of Part II only) lists items by subject.
- Reference Guide sections 1.1 and 2.1 provide lists of commands and subroutines, respectively, by functional category.

Part III is a reference guide for subsystem writers. It is of interest to compiler writers and writers of sophisticated subsystems. It documents user-accessible modules which allow a user to bypass standard Multics facilities. The interfaces thus documented are a level deeper into the system than those required by the casual user.

Examples of specialized subsystems for which construction would require reference to Part III are:

1) a subsystem which precisely imitates the command environment of some system other than Multics (e.g., an imitation of the Dartmouth Time-Sharing System);

2) a subsystem which is intended to enforce restrictions on the services available to a set of users (e.g., an APL-only subsystem for use in an academic class);

3) a subsystem which is protecting some kind of information in a way not easily expressible with ordinary access control lists (e.g., a proprietary linear programming system, or an administrative data base system which permits access only to program-defined aggregated information such as averages and correlations).

Each of the three parts of the MPM has its own table of contents and is updated separately, by adding and replacing individual sections. Each section is separately dated, both on the section itself, and in the appropriate table of contents. The title page and table of contents are replaced as part of each update, so one can quickly determine if his manual is properly up-to-date. The Multics on-line "message of the day" or local installation bulletins should provide notice of availability of new updates. In addition, the Multics command "help mpm" provides on-line information about known errors and the latest MPM update level.

In addition to this manual, users who will write programs for Multics will need a manual giving specific details of the language they will use; such manuals are currently available for PL/I, FORTRAN, and BASIC. A separate, specialized supplement to the MPM is also provided for users of graphic displays. The bibliography at the end of Part I, Chapter 1, describes these and other references in more detail.

Multics provides the ability for a local installation to develop an installation-maintained or author-maintained library of commands and subroutines which are tailored to local needs. The installation may also document these facilities in the same format as used in the MPM; the user can then interfile these locally provided write-ups in the command and subroutine sections of his MPM.

Finally, access to Multics requires authorization. The prospective user must negotiate with the administration of his local installation for permission to use the system. The installation may find it useful to provide the new user with a documentation kit describing available documents, telephone numbers, operational schedules, consulting services, and other local conventions.

CONTENTS

September 30, 1973

CHAPTER 1

HIGHLIGHTS OF THE MULTICS SYSTEM

September 20, 1973

Introduction

Multics (from Multiplexed Information and Computing Service) is the name of a new, general-purpose computer system developed by the Computer Systems Research Division of M.I.T. Project MAC, in cooperation with Honeywell Information Systems (formerly the General Electric Company computer department) and the Bell Telephone Laboratories. This system is designed to be a "computer utility", extending the basic concepts and philosophy of earlier time-sharing systems in many directions. Multics was implemented initially on the Honeywell 645 computer system, an enhanced relative of the Honeywell 635 computer. It currently uses a Honeywell 6180 computer system.

The Goals

The goals of the Multics system were set out in 1965 in a paper by Corbató and Vyssotsky. While those goals have been met only partially in some cases, most of the original plans have been realized. The 1965 paper described those goals as follows:*

"One of the overall design goals of Multics is to create a computing system which is capable of meeting almost all of the present and near future requirements of a large computer utility. Such systems must run continuously and reliably 7 days a week, 24 hours a day, in a way similar to telephone or power systems, and must be capable of meeting wide service demands: from multiple man-machine interaction to the sequential processing of absentee user jobs; from the use of the system with dedicated languages and subsystems to the programming of the system itself; and from

---

* From a modified version of: Corbató, F.J., and Vyssotsky, V.A., "Introduction and Overview of the Multics System", AFIPS Conf. Proc. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 185-196. Copyright 1965 by AFIPS Press, reprinted by permission.

centralized bulk card, tape, and printer facilities to remotely located terminals. Such information processing and communication systems are believed to be essential for the future growth of computer use in business, in industry, in government and in scientific laboratories, as well as stimulating applications which would otherwise be untried.

"Because the system must ultimately be comprehensive and able to adapt to unknown future requirements, its framework must be general, and capable of evolving with time. As brought out in the sequel, this need for an evolutionary framework influences and contributes to much of the system design and is a major reason why most of the programming of the system has been done in a subset of the PL/I language. Because the PL/I language is largely machine-independent (e.g., data descriptions refer to logical items, not physical words), the system should also be. Specifically, it is hoped that future hardware improvements will not make system and user programs obsolete and that implementation of the entire system on other suitable computers will require only a moderate amount of additional programming....

"As computers have matured during the last two decades from curiosities to calculating machines to information processors, access to them by users has not improved, and, in the case of most large machines, has retrogressed. Principally for economic reasons, batch processing of computer jobs has been developed and is currently practiced by most large computer installations, and the concomitant isolation of the user from elementary cause-and-effect relationships has been either reluctantly endured or rationalized. For several years a solution has been proposed to the access problem. This solution, usually called time-sharing, is basically the rapid time-division multiplexing of a central processor unit among the jobs of several users, each on-line at a typewriter-like terminal. The rapid switching of the processor unit among user programs is, of course, nothing but a particular form of multiprogramming....

"The impetus for time-sharing first arose from professional programmers because of their constant frustration in debugging programs at batch processing installations. Thus, the original goal was to time-share computers to allow simultaneous access by several persons while giving to each of them the illusion of having the whole machine at his disposal. This goal led to the development of the Compatible Time-Sharing System (CTSS) at M.I.T. Project MAC. However, at Project MAC it has turned out that simultaneous access to the machine, while obviously necessary to the objective, has not been the major ensuing benefit. Rather, it is the availability at one's fingertips of facilities for editing, compiling, debugging, and running programs in one continuous interactive session that has had the greatest effect on programming. Professional programmers are encouraged to be more imaginative in their work and to investigate new programming techniques and new problem approaches because of the much smaller penalty for failure. But, the most significant effect that CTSS has had on the M.I.T. community is seen in the achievements of persons for whom computers are tools for other objectives. The availability of CTSS not only has changed the way problems are attacked, but has caused important research to be undertaken that otherwise would not have been done. As a consequence, the objective of the current and future development of time-sharing extends beyond the improvement of computational facilities with respect to traditional computer applications. Rather, it is the on-line use of computers for new purposes and in new fields which provides the challenge and the motivation to the system designer. In other words, the major goal is to provide suitable tools for what is currently being called machine-aided cognition.

"More specifically, the importance of a multiple-access system operated as a computer utility is that it allows a vast enlargement of the scope of computer-based activities, which can, in turn, stimulate a corresponding enrichment of many areas of our society. Over ten years of experience indicates that continuous operation in a utility-like manner, with flexible remote access, encourages users to view the system as a thinking tool in their daily intellectual work. Mechanistically, the qualitative change from the past results from the drastic improvement in access time and convenience. Objectively, the change lies in the user's ability to control and affect interactively the course of a process whether it involves numerical computation or manipulation of symbols. Thus, parameter studies are more intelligently guided; new problem-oriented languages and subsystems are developed to exploit the interactive capability; many complex analytical problems, as in magnetohydrodynamics, which have been too cumbersome to be tackled in the past, are now being successfully pursued; even more, new, imaginative approaches to basic research have been developed as in the decoding of protein structures. These are examples taken from an academic environment; the effect of multiple-access systems on business and industrial organizations can be equally dramatic. It is with such new applications in mind that the Multics system has been developed. Not that the traditional uses of computers are being disregarded: rather, these traditional needs are viewed as a subset of the broader, more demanding, new requirements.

"To meet the above objectives, issues such as response time, convenience of manipulating data and programs, ease of controlling processes during execution, and, above all, protection of private information and isolation of independent processes, become of critical importance. These issues demand departures from traditional computer systems. While these departures are deemed to be desirable with respect to traditional computer applications, they are essential for rapid man-machine interaction.

System Requirements

"In the early days of computer design, there was the concept of a single program on which a single processor computed for long periods of time with almost no interaction with the outside world. Today such a view is considered incomplete. The effective boundaries of an information processing system extend beyond the processor, beyond the card reader and printer, and even beyond the typing of input and the printing of output. In fact, they encompass the goals of many people. To better understand the effect of this broadened design scope, it is helpful to examine several phenomena characteristic of large, service-oriented computer installations.

"First, there are incentives for any organization to have the biggest possible computer system that it can afford. It is usually only on the biggest computers that there are elaborate programming systems, compilers, and features which make a computer "powerful". This results partly because it is more difficult to prepare system programs for smaller computers when limited by speed or memory size, and partly because large systems involve more persons and, hence, permit more attention to be given to system programs. Moreover, by combining resources in a single computer system rather than in several, bulk economies and therefore lower computing costs can be achieved. Finally, as a practical matter, considerations of floor space, management efficiency, and operating personnel provide a strong incentive for centralizing computer facilities in a single large installation.

"Second, the capacity of a contemporary computer installation, regardless of the sector of applications it serves, must be capable of growing to meet continuously increasing demand. A doubling of demand every two years is not uncommon. Multiple-access computers promise to accelerate this growth further since they allow a man-machine interaction rate which is faster by at least two orders of magnitude than other types of computing systems. Present indications are that multiple-access systems for only a few hundred users can generate a demand for computation exceeding the capacity of the fastest existing single processor system. Since the speed of light, the physical sizes of computer components, and the speeds of memories are intrinsic limitations on the speed of any single processor, it is clear that systems with multiple processors and multiple memory units are needed to provide greater capacity. This is not to say that fast processor units are undesirable, but that extreme system complexity to enhance this single parameter among many appears neither wise nor economic.

"Third, computers are no longer a luxury used when and if available, but are primary working tools in business, government, and research laboratories. The more reliable computers become, the more their availability is depended upon. A system structure including pools of functionally identical units (processors, memory modules, input/output controllers, etc.) can provide continuous service without significant interruption for equipment maintenance, as well as provide growth capability through the addition of appropriate units.

"Fourth, user programs, especially in a time-sharing system, interact frequently with secondary storage devices and terminals. This communication traffic produces a need for multiprogramming to avoid wasting main processor time while an input/output request is being completed. It is important to note that an individual user is ordinarily not in a position to do an adequate job of multiprogramming since his program lacks proper balance, and he probably lacks the necessary dynamic information, ingenuity, or patience.

"Finally, as noted earlier, the value of a time-sharing system lies not only in providing, in effect, a private computer to a number of people simultaneously, but, above all, in the services that the system places at the fingertips of the users. Moreover, the effectiveness of a system increases as user-developed facilities are shared by other users. This increased effectiveness because of sharing is due not only to the reduced demands for core and secondary memory, but also to the cross-fertilization of user ideas. Thus, a major goal of the present effort is to provide multiple access to a growing and potentially vast structure of shared data and shared program procedures. In fact, the achievement of multiple access to the computer processors should be viewed as but a necessary subgoal of this broader objective. Thus, the primary and secondary memories where programs reside play a central role in the hardware organization, and the presence of independent communication paths between memories, processors, and terminals is of critical importance.

"From the above it can be seen that the system requirements of a computer installation are not for a single program on a single computer, but, rather, for a large system of many components serving a community of users. Moreover, each user of the system asynchronously initiates jobs of arbitrary and indeterminate duration which subdivide into sequences of processor and input/output tasks. It is out of this seemingly chaotic, random environment that one arrives at a utility-like view of a computing system. For instead of chaos, one can average over the different user requests to achieve high utilization of all resources. The task of multiprogramming required to do this need only be organized once in a central supervisor program. Each user thus enjoys the benefit of efficiency without having to average the demands of his own particular program.

"With the above view of computer use, where tasks start and stop every few milliseconds, and where the memory requirements of tasks grow and shrink, it is apparent that one of the major jobs of the supervisor program (i.e., monitor, executive, etc.) is the allocation and scheduling of computer resources. The general strategy is clear. Each user's job is subdivided into tasks, usually as the job proceeds, each of which is placed in an appropriate queue (i.e., for a processor or an input/output controller). Processors or input/output controllers are, in turn, assigned new tasks as they either complete or are removed from old tasks. All processors are treated equivalently in an anonymous pool and are assigned to tasks as needed. In particular, the supervisor does not have a special processor. Further, processors can be added or deleted without significant change in either the user or system programs. Similarly, input/output controllers are directed from queues independently of any particular processor. Again, as with the processors, one can add or delete input/output capacity according to system load without significant reprogramming required.

The Multics System

"The overall design goal of the Multics system is to create a computing system which is capable of comprehensively meeting almost all of the present and near future requirements of a large computer service installation. It is not expected that the initial system, although useful, will reach the objective; rather, the system will evolve with time in a general framework which permits continual growth to meet unknown future requirements. The use of the PL/I language will allow major system software changes to be developed on a schedule separate from that of hardware changes. Since most organizations can no longer afford to overlap old and new equipment during changes, and since software development is at best difficult to schedule, this relative machine-independence should be a major asset."

Overview of Multics Capabilities

An ability to share data contained within the framework of a general purpose time-sharing system is a unique feature of Multics, and is directly applicable to administrative problems, research requiring a multi-user accessible data base, and general application of the computer to very complicated research problems. The attention paid to mechanisms to provide and control privacy is of direct interest for several of the same applications as well as, for example, medical data. Multics can thus be a valuable tool which provides opportunities for important new research in these areas.

Multics offers a number of additional capabilities which go well beyond those provided by many other systems. Those which are most significant from the user's point of view are described here. Perhaps the most interesting aspect of all is that a single system encompasses all of these capabilities simultaneously.

1. The ability to be a small user of Multics.

   An underlying consideration throughout the Multics design has been that the simple user should not pay a noticeable extra price for a system which also accomodates the sophisticated user. For example, a student can be handed a limited set of tools, can do limited work (perhaps debugging and running small BASIC programs), and expect to receive a bill for resource usage which is proportional to the limited work done. If all users are small, of course, the number of users can be increased in proportion to their smallness. As an administrative aid, facilities are provided so that one can restrict any particular user to a specific set of tools and thereby limit his ability to use up resources.

2. The ability to control sharing of information.

   There are a variety of applications for a computer system which involve building up a base of information which is to be shared among several individuals. Multics provides facilities in two directions.

   Sharing:

   - Links to other users' programs and data.
   - Ability to move one's base of operation into another user's directory (with his permission).
   - Direct access with uniform conventions to any information stored in the system.
   - Ability for two or more users to share a single copy of a program or data in core memory.

   Control:

   - Ability to specify precisely to whom, and with what access mode (e.g., read, write, and execute permissions are separate and per-user) a piece of data or the entire contents of a subdirectory are available.
   - Ability to revoke access at any time.
   - Ability, using the Multics protection ring structure, to force access to a data base to be only via a program supplied by the data base owner. This facility may be used to allow access to aggregate information, such as averages or counts, or specified data entries, without simultaneously giving access to the entire file of raw data, which may be confidential. There are a large number of

potential administrative applications of this feature, and as far as is known, Multics is the only general-purpose system which provides it.

3. The virtual memory approach.

In the opposite direction of the little user is the person with a difficult research problem requiring a very large addressable memory. The Multics storage system, with the aid of a high-performance paging system, provides this facility in what is often called a virtual memory of an extent limited only by the total of secondary storage devices (drums, disks, etc.) attached to the system. An interesting property of the Multics implementation is that a procedure may be written to operate in a very large virtual memory, but primary memory resources are used only for those parts of the virtual memory actually touched by the program on that execution, and disk and drum resources are used only for those parts of the memory which actually contain data. Another very useful property from a programmer's point of view is that information stored in the storage system is directly accessible to his program by a virtual memory address. This property eliminates the need for explicitly programmed overlays, chain links, or memory loads, and also reduces the number of explicitly programmed input and output operations. The Multics storage system takes on the responsibility for safekeeping of all information placed there by the user. It therefore automatically maintains tape copies of all information which has remained in the system for more than an hour. These tapes can be used to reload any user information lost or damaged as a result of hardware or software failures, and may also be used to retrieve individual items damaged by a user's own blunder.

Each user has an administratively set quota of space which limits the amount of storage he can use, although he may purchase as large an amount of space as he would like. Additional disk storage can be added to the system in large quantities if necessary.

4. The option of dynamic linking.

In constructing a program or system of programs, it is frequently convenient to begin testing certain features of one program before having written another program which is needed for some cases. Dynamic linking allows the execution of the first program to begin, and a search for the second program is undertaken only if and when it is actually called by the first one. This feature also allows a user to freely include in his program a conditional call out to a large and sophisticated error diagnostic program, secure in the knowledge that in all those executions of his program which do not encounter the error, he will not pay the cost of locating, linking, and mapping into his virtual memory the error diagnosis package. It also allows a user borrowing a program to provide a substitute for any subroutine called by that program when he uses it, since he has control over where the system looks to find missing subroutines. In those cases where subroutine A calls subroutine B every time, there is, of course, no need to use dynamic linking (and the implied library search), so facilities are provided to bind A and B together prior to execution.

5. Configuration flexibility.

   An important aspect of the Multics design is that it is actually difficult for a user to write a program which will stop working correctly if the hardware configuration is changed. In response to changing system-wide needs, the amount of primary memory, the number of central processors, the amount and nature of secondary storage (disks, drums, etc.), and the type of interactive typewriter terminals may change with time over a range of 2 or 3 to 1, but users do not normally need to change their programs to keep up with the hardware. The system itself adapts to changes in the number of processor or memory boxes dynamically, that is, while users are logged in. Most other configuration changes (e.g., the addition of disk storage units) require that the system be reinitialized, an operation which takes a few minutes.

6. The human interface.

   Experience has proven that ease of use of a time-sharing system is considerably more sensitive to human engineering than is a batch processing system. The Multics command language has been designed with this in mind. Features such as universal use of a character set with both upper and lower case letters in it, and allowing names of objects to be 32 characters long, are examples of the little things which allow the nonspecialist to feel that he does not have to discover a secret code in order to be an effective user of the system. In a similar vein, a hierarchial storage system provides a very useful organization and bookkeeping aid, so that a user need keep immediately at hand only those things he is working with at the moment. Such a facility is of great assistance when attacking complicated or intricately structured problems.

Languages

Multics provides two primary user languages: PL/I and FORTRAN IV. The FORTRAN compiler is fairly standard. It is supported by the usual library of math routines and formatted input/output facilities. Its primary use is for translation of already written programs which have been imported from other computer systems.

The Multics PL/I compiler is quite interesting because it offers a very full selection of language facilities, over 300 helpful error diagnostics, and the ability to get at the advanced features of Multics, all at reasonable cost. For these reasons, as well as the availability of PL/I on other computer systems, it is the recommended language for subsystem implementers and general research users needing an expressive language. If is worth noting that the system itself is written mostly in the PL/I language.

Other languages available on Multics are:

BASIC - A translator and editor subsystem for the BASIC language, developed at Dartmouth College. A limited Multics service is available which restricts the user to just this subsystem, if desired. The BASIC subsystem is also available to regular Multics users.

APL - A powerful and popular interpretive language developed by Kenneth Iverson. The Multics implementation very closely imitates Iverson's, with the exception that an effectively unlimited workspace size is available.

LISP - Both an interpreter and a compiler are available for this list processing language often used in artificial intelligence applications. The Multics implementation of the MACLISP dialect of LISP contains useful and sophisticated features not available in most other dialects of LISP. Among these are debugging tools and the ability to modify or program parts of the interpreter. The latter makes it an easily extensible language. Another interesting feature of the Multics implementation is the very large structure space provided by the virtual memory.

ALM - A machine language assembler for the Honeywell 6180 computer. (It is not recommended for general use; it is slow and the machine language is very difficult.)

QEDX - A programmable editor which qualifies as a minor interpretive language.

All of the above languages translate a source program which has been previously placed in the storage system. Input and editing of source text is done with one of the available text editors, edm or qedx. Although interactive, line-by-line syntax checking languages are easily implemented in the Multics environment, none are currently available.

A source language debugging system, named debug, provides the ability to inspect variables and set break points in terms of the PL/I or FORTRAN program being debugged. It also has a variety of features to allow inspection of all aspects of the Multics execution environment.

A Multics Bibliography

A. Manuals which are available through Honeywell.

1. Multics Programmers' Manual (Order Numbers AG90, AG91, AG92, AG93 and AK92). An updateable reference manual in five volumes. Volume I is an introduction to the Multics programming environment and includes sample terminal sessions and annotated Multics programs. Volume II contains reference material on the overall mechanics, conventions and usage of the system. Volumes III and IV are alphabetically organized lists of standard Multics commands and subroutines, respectively, giving details of the calling sequence and usage of each. Volume V provides reference material and descriptions of commands and subroutines which are of interest primarily to compiler writers and subsystem writers.

2. The Multics PL/I Language (Order Number AG94). A reference manual which specifies precisely the PL/I language used on Multics.

3. The Multics Virtual Memory (Order Number AG95). A collection of three technical papers on the hardware and software used to implement the virtual memory and program protection features of Multics.

4. Multics Project Administrators' Manual, preliminary edition (Order Number AK51). A reference manual for project adminstators describing commands and subroutines which may be used to specify certain features of Multics to the members of a project.

5. Multics System Administrators' Manual (Order Number AK50). A reference manual for system administrators of a Multics installation describing commands and subroutines which may be used to control various system parameters.

6. The APL User's Guide (Order Number AK95). A manual for beginning and advanced APL users describing the use of Multics APL.

B. Manuals which may be examined in the M.I.T. Project MAC or Information Processing Center Document Rooms. These manuals are not otherwise available.

1. Multics System Programmers' Manual. In principle, a complete reference manual describing how the system works inside. In fact, this document contains many sections which are inconsistent, inaccurate, or obsolete; it is in need of much upgrading. However, its overview sections are generally accurate and valuable if insight into the internal organization is desired.

2. System Programmers' Supplement to the Multics Programmers' Manual. This updateable reference manual, in the same format as the Multics Programmers' Manual, provides calling sequences of every system module.

3. Graphic Users' Supplement to the Multics Programmers' Manual. In the same format as the Multics Programmers' Manual, this supplement gathers in one place descriptions of the Multics Graphics System, and the commands and subroutines needed to use it.

4. A User's Guide to Multics FORTRAN. A document which provides the prospective Multics FORTRAN user with sufficient information to enable him to create and execute FORTRAN programs on Multics. It contains a complete definition of the Multics FORTRAN language as well as a description of the FORTRAN command and error messages. It also describes how to communicate with non-FORTRAN programs, and discusses some of the fundamental characteristics of Multics which affect the FORTRAN user.

5. EPLBSA Programmer's Reference Handbook, by D. J. Riesenberg. A manual describing the assembly (machine) language for the Honeywell 645 computer. The language has been renamed ALM since the publication of this manual. (Needed only by programmers with some special reason to use 645 machine language.)

6. Honeywell 645 Processor Manual. A hardware description including opcodes, addressing modifiers, etc. Of interest only to dedicated machine language programmers.

C. Books about Multics.

1. The Multics System: An Examination of its Structure, by E. I. Organick. A hard cover book describing in some detail how Multics works. The description is from the point of view of a programmer developing a large

program or subsystem, who wishes to gain the extra insight to help him intelligently choose among available alternatives of his implementation. M.I.T. Press, Cambridge, Mass., 1972. 392 pages.

2. Time-Sharing System Concepts, by R. Watson. A book comparing many aspects of the planned implementation of Multics (as originally described in the Multics System Programmers' Manual) with the SDS-940 time-sharing system developed at the University of California at Berkeley. Although the actually implemented Multics differs greatly from the one described in this book, much can be learned from it about the problems of large-scale system organization. McGraw Hill, New York, 1970. 270 pages.

D. Technical Papers About Multics.

1. Corbató, F. J., and Vyssotsky, V. A., "Introduction and Overview of the Multics System", AFIPS Conf. Proc. 27 (1965 FJCC), pp. 185-196.

2. Glaser, E. L., et al., "System Design of a Computer for Time-Sharing Application", AFIPS Conf. Proc. 27 (1965 FJCC), pp. 197-202.

3. Vyssotsky, V. A., et al., "Structure of the Multics Supervisor", AFIPS Conf. Proc. 27 (1965 FJCC), pp. 203-212.

4. Daley, R. C., and Neumann, P. G., "A General-Purpose File System for Secondary Storage", AFIPS Conf. Proc. 27 (1965 FJCC), pp. 213-229.

5. Ossanna, J. F., et al., "Communication and Input/Output Switching in a Multiplex Computing System", AFIPS Conf. Proc. 27 (1965 FJCC), pp. 231-241.

6. David, E. E., Jr., and Fano, R. M., "Some Thoughts About the Social Implications of Accessible Computing", AFIPS Conf. Proc. 27 (1965 FJCC), pp. 243-247.

7. Glaser, E.L., "A Brief Description of the Privacy Measures in the Multics Operating System", AFIPS Conf. Proc. 31 (1967 FJCC), pp. 303-304.

8. Bensoussan, A., Clingen, C.T., and Daley, R.C., "The Multics Virtual Memory: Concepts and Design", Comm. ACM 15, 5 (May, 1972), pp. 308-318.

9. Clingen, C. T., "Program Naming Problems in a Shared Tree-Structured Hierarchy", NATO Science Committee Conference on Techniques in Software Engineering, 1 (October 27-31, 1969), Rome, Italy.

10. Graham, R.M., "Protection in an Information Processing Utility", Comm. ACM 11, 5 (May, 1968), pp. 365-369.

11. Daley, R. C., and Dennis, J. B., "Virtual Memory, Processes, and Sharing in Multics", Comm. ACM 11, 5 (May, 1968), pp. 306-312.

12. Corbató, F. J., and Saltzer, J. H., "Some Considerations of Supervisor Program Design for Multiplexed Computer Systems", Proc. IFIP Conf. 1968 Invited Papers, pp. 66-72.

13. Corbató, F. J., "PL/I as a Tool for System Programming", Datamation 15, 6 (May, 1969), pp. 68-76.

14. Corbató, F. J., "A Paging Experiment with the Multics System", In Honor of P. M. Morse, M.I.T. Press, Cambridge, Massachusetts, 1969, pp. 217-228.

15. Saltzer, J.H., and Gintell, J.W., "The Instrumentation of Multics", Comm. ACM 13, 8 (August, 1970), pp. 495-600.

16. Spier, M. J., and Organick, E. I., "The Multics Inter-Process Communication Facility", ACM Second Symposium on Operating System Principles (October 20-22, 1969), Princeton University, pp. 83-91.

17. Freiburghouse, R. A., "The Multics PL/I Compiler", AFIPS Conf. Proc. 35 (1969), AFIPS Press, 1969, pp. 187-199.

18. Grochow, J. M., "Real-Time Graphic Display of Time-Sharing System Operating Characteristics", AFIPS Conf. Proc. 35 (1969 FJCC), AFIPS Press, 1969, pp. 379-385.

19. Saltzer, J. H., and Ossanna J. F., "Remote Terminal Character Stream Processing in Multics", AFIPS Conf. Proc. 36 (1970 SJCC), AFIPS Press, 1970, pp. 621-627.

20. Ossanna, J. F., and Saltzer, J. H., "Technical and Human Engineering Problems in Connecting Terminals to a Time-Sharing System", AFIPS Conf. Proc. 37 (1970 FJCC), AFIPS Press, 1970, pp. 355-362.

21. Clark, D. D., Graham, R. M., Saltzer, J. H., and Schroeder, M. D., "Classroom Information and Computing Service", M.I.T. Project MAC Technical Report TR-80, (January 11, 1971).

22. Schroeder, M. D., "Performance of the GE-645 Associative Memory While Multics is in Operation", ACM Workshop on System Performance Evaluation (April, 1971), pp. 227-245.

23. Schroeder, M.D., and Saltzer, J.H., "A Hardware Architecture for Implementing Protection Rings", Comm. ACM 15, 3 (March, 1972), pp. 157-170.

24. Feiertag, R. J., and Organick, E. I., "The Multics Input/Output System", ACM Third Symposium on Operating Systems Principles (October 18-20, 1971), Palo Alto, California.

25. Sekino, A., "Response Time Distribution of Multiprogrammed Time-Shared Computer Systems", Sixth Annual Princeton Conference on Information Sciences and Systems, March 23-24, 1972, Princeton, N.J.

26. Corbató, F. J., Saltzer, J. H., and Clingen, C. T., "Multics--The First Seven Years", AFIPS Conf. Proc. 40 (1972 SJCC) AFIPS Press, 1972. pp. 571-583.

27. Wolman, B.L., "Debugging PL/I Programs in the Multics Environment," AFIPS Conf. proc. 41, Part 1, (1972 FJCC), AFIPS Press, 1972, pp. 507-514.

28. Saltzer, J.H., "Protection and Control of Information Sharing in Multics", ACM Fourth Symposium on Operating System Principles (October, 1973), Yorktown Heights, New York.

29. Scheffler, L., "Optimal Folding of a Paging Drum in a Three-level Memory", ACM Fourth Symposium on Operating System Principles (October, 1973), Yorktown Heights, New York.

E. M.I.T. Theses Related to Multics. Those followed by MAC-TR numbers are also printed as M.I.T. Project MAC technical reports, and are available from the National Technical Information Service in Springfield, Virginia.

1. Saltzer, J. H., "Traffic Control in a Multiplexed Computer System", Sc.D., 1966. (MAC-TR-30)

2. Rappaport, R., "Implementing Multi-Process Primitives in a Multiplexed Computer System", S.M., 1968. (MAC-TR-55)

3. Deitel, H., "Absentee Computations in a Multiple-Access Computer System", S.M., 1968. (MAC-TR-52)

4. Greenbaum, J., "A Simulator of Multiple Interactive Users to Drive a Time-Shared Computer System", S.M., 1968. (MAC-TR-58)

5. Grochow, J. M., "The Graphic Display as an Aid in the Monitoring of a Time-Shared Computer System", S.M., 1968. (MAC-TR-54)

6. Schroeder, M. D., "Classroom Model of an Information and Computing Service", S.M., 1969.

7. Frankston, R., "A Limited Service System on Multics", S.B., June, 1970.

8. Schell, R. R., "Dynamic Reconfiguration in a Modular Computer System", Ph.D., June, 1971. (MAC-TR-86)

9. Sekino, A., "Performance Evaluation of Multiprogrammed Time-Shared Computer Systems", Ph.D., August, 1972. (MAC-TR-103)

10. Schroeder, M.D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility", Ph.D., September, 1972. (MAC-TR-104)

11. Reed, D., "Estimation of Primary Memory Requirements of Processes on Multics", S.B., June, 1973.

12. Stern, J., "Automatic File Backup in a Computer Utility", S.M., September, 1973. (MAC-TR-116)

13. Rotenberg, L., "Making Computers Keep Secrets", PH.D., September, 1973. (MAC-TR-115)

14. Clark, D., "Input/Output in a Virtual Memory Computer System", Ph.D., September, 1973. (MAC-TR-117)

15. Gumpertz, R., "The Design and Fabrication of an ARPA Network Interface", S.B., September, 1973.

CHAPTER 2

INTRODUCTION TO THE CONCEPTS OF MULTICS

September 20, 1973

The following pages contain reprints of eight technical papers about Multics. Although these papers were written individually for conferences and technical journals, as a group they provide an in-depth introduction to most of the major concepts of the Multics system. The reader should be warned that the earliest of these papers was written six years before the latest. As a result, he will notice minor differences in terminology and emphasis, reflecting the gradually increasing experience both in using and explaining ideas which were first introduced by Multics. In addition, these papers should be taken as background explanations of why Multics is designed the way it is, rather than as a reference to the way it currently works. Some ideas suggested in these papers have not yet been implemented in the actual system, or having been implemented and found wanting, have been discarded. Parts II and III of the Multics Programmers' Manual provide current descriptions of the user interfaces which are actually implemented in Multics, and should be used as reference for all programming. On the other hand, much of that reference guide merely tells how, without explaining why, which is the purpose of this chapter.

The reader who is interested in a greater depth of detail about Multics may wish to consult the book _The Multics System: An Examination of Its Structure_, by Elliott I. Organick (MIT Press, 1972). That book provides a deep and authoritative look at the implementation of many of the parts of the Multics system. In addition, the bibliography at the end of MPM Introduction Chapter One provides a list of other specialized technical papers and academic theses related to Multics.

Finally, the reader who wishes only to use the Multics system will probably want to only skim this chapter to see what kinds of ideas are discussed here. It is _not_ necessary to comprehend Chapter Two in order to begin using Multics. The concepts provided here are background in nature, and are probably most useful to a reader contemplating an unusual application of the system. For an introduction on how to use and program for Multics, one should move on to Chapters Three and Four of the manual.

Multics -- The First Seven Years

by F.J. Corbató, J.H. Saltzer, and C.T. Clingen. Reprinted from AFIPS Conference Proceedings 40, AFIPS press, 1972, pp. 571-583, with permission. Copyright 1972 by AFIPS Press.

This overview chapter is one of the most recent, and is therefore quite up-to-date in terminology and method of description. Although it does not explore any single technical topic in depth, it includes a wide range of facts about the Multics system, and provides a perspective on which aspects of the system are especially significant. [illegible], this paper provides a summary of the [illegible] development which Multics had attained as of [illegible], 1972.

Multics—The first seven years*

Reprinted from —
AFIPS — Conference Proceedings
Volume 40
© AFIPS PRESS
Montvale, N. J. 07645

by F. J. CORBATÓ and J. H. SALTZER

Massachusetts Institute of Technology
Cambridge, Massachusetts

and

C. T. CLINGEN

Honeywell Information Systems Inc.
Cambridge, Massachusetts

INTRODUCTION

In 1964, following implementation of the Compatible Time-Sharing System (CTSS)[1,2] serious planning began on the development of a new computer system specifically organized as a prototype of a computer utility. The plans and aspirations for this system, called Multics (for **Multi**plexed **I**nformation and **C**omputing **S**ervice), were described in a set of six papers presented at the 1965 Fall Joint Computer Conference.[3–8] The development of the system was undertaken as a cooperative effort involving the Bell Telephone Laboratories (from 1965 to 1969), the computer department of the General Electric Company,* and Project MAC of M.I.T.

Implicit in the 1965 papers was the expectation that there should be a later examination of the development effort. From the present vantage point, however, it is clear that a definitive examination cannot be presented in a single paper. As a result, the present paper discusses only some of the many possible topics. First we review the goals, history and current status of the Multics project. This review is followed by a brief description of the appearance of the Multics system to its various classes of users. Finally several topics are given which represent some of the research insights which have come out of the development activities. This organization has been chosen in order to emphasize those aspects of software systems having the goals of a computer utility which we feel to be of special interest. We do not attempt detailed discussion of the organization of Multics; that is the purpose of specialized technical books and papers.*

* Work reported herein was sponsored (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under office of Naval Research Contract Number N00014-70-A-0362-0001. Reproduction is permitted for any purpose of the United States Government.

* Subsequently acquired by Honeywell Information Systems Inc.

GOALS

The goals of the computer utility, although stated at length in the 1965 papers, deserve a brief review. By a computer utility it was meant that one had a community computer facility with:

(1) Convenient remote terminal access as the normal mode of system usage;
(2) A view of continuous operation analogous to that of the electric power and telephone companies;
(3) A wide range of capacity to allow growth or contraction without either system or user reorganization;
(4) An internal file system so reliable that users trust their only copy of programs and data to be stored in it;
(5) Sufficient control of access to allow selective sharing of information;
(6) The ability to structure hierarchically both the logical storage of information as well as the administration of the system;
(7) The capability of serving large and small users without inefficiency to either;
(8) The ability to support different programming environments and human interfaces within a single system;

* For example, the essential mechanisms for much of the Multics system are given in books by Organick[9] and Watson.[10]

(9) The flexibility and generality of system organization required for evolution through successive waves of technological improvements and the inevitable growth of user expectations.

In an absolute sense the above goals are extremely difficult to achieve. Nevertheless, it is our belief that Multics, as it now exists, has made substantial progress toward achieving each of the nine goals.* Most importantly, none of these goals had to be compromised in any important way.

HISTORY OF THE DEVELOPMENT

As previously mentioned, the Multics project got under way in the Fall of 1964. The computer equipment to be used was a modified General Electric 635 which was later named the 645. The most significant changes made were in the processor addressing and access control logic where paging and segmentation were introduced. A completely new Generalized Input/Output Controller was designed and implemented to accommodate the varied needs of devices such as disks, tapes and teletypewriters without presenting an excessive interrupt burden to the processors. To handle the expected paging traffic, a 4-million word (36-bit) high-performance drum system with hardware queueing was developed. The design specifications for these items were completed by Fall 1965, and the equipment became available for software development in early 1967.

Software preparation underwent several phases. The first phase was the development and blocking out of major ideas, followed by the writing of detailed program module specifications. The resulting 3,000 typewritten pages formed the Multics System Programmers' Manual and served as the starting point for all programming. Furthermore, the software designers were expected to implement their own designs. As a general policy PL/I was used as the system programming language wherever possible to maximize lucidity and maintainability of the system.[14,15] This policy also increased the effectiveness of system programmers by allowing each one to keep more of the system within his grasp.

The second major phase of software development, well under way by early 1967, was that of module implementation and unit checkout followed by merging into larger aggregates for integrated testing. Up to then most software and hardware difficulties had been anticipated on the basis of previous experience. But what gradually became apparent as the module integration continued was that there were gross discrepancies between actual and expected performance of the various logical execution paths throughout the software. The result was that an unanticipated phase of design iterations was necessary. These design iterations did not mean that major portions of the system were scrapped without being used. On the contrary, until their replacements could be implemented, often months later, they were crucially necessary to allow the testing and evaluation of the other portions of the system. The cause of the required redesigns was rarely "bad coding" since most of the system programmers were well above average ability. Moreover the redesigns did not mean that the goals of the project were compromised. Rather three recurrent phenomena were observed: (1) typically, specifications representing less-important features were found to be introducing much of the complexity, (2) the initial choice of modularity and interfacing between modules was sometimes awkward and (3) it was rediscovered that the most important property of algorithms is simplicity rather than special mechanisms for unusual cases.*

The reason for bringing out in detail the above design iteration experience is that frequently the planning of large software projects still does not properly take the need for continuing iteration into account. And yet we believe that design iterations are a required activity on any large scale system which attempts to break new conceptual ground such that individual programmers cannot comprehend the entire system in detail. For when new ground is broken, it is usually impossible to deduce the consequent system behavior except by experimental operation. Simulation is not particularly effective when the system concepts and user behavior are new. Unfortunately, one does not understand the system well enough to simplify it correctly and thereby obtain a manageable model which requires less effort to implement than the system itself. Instead one must develop a different view:

(1) The initial program version of a module should be viewed only as the first complete specification of the module and should be subject to design review *before* being debugged or checked out.
(2) Module design and implementation should be based upon an assumption of periodic evaluation, redesign, and evolution.

In retrospect, the design iteration effect was apparent

* To the best of our knowledge, the only other attempt to comprehensively attack all of these goals simultaneously is the TSS/360 project at IBM.[11,12,13]

* "In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away . . . "
—Antoine de Saint-Exupéry, *Wind, Sand and Stars* Quoted with permission of Harcourt Brace Jovanovich, Inc.

even in the development of the earlier Compatible Time-Sharing System (CTSS) when a second file system with many functional improvements turned out to have poor performance when initially installed. A hasty design iteration succeeded in rectifying the matter but the episode at the time was viewed as an anomaly perhaps due to inadequate technical review of individual programming efforts.

CURRENT STATUS

In spite of the unexpected design iteration phase, the Multics system became sufficiently effective by late 1968 to allow system programmers to use the system while still developing it. By October 1969, the system was made available for general use on a "cost-recovery" charging basis similar to that used for other major computation facilities at M.I.T. Multics is now the most widely used time-sharing system at M.I.T., supporting a user community of some 500 registered subscribers. The system is currently operated for users 22 hours per day, 7 days per week. For at least eight hours each day the system operates with two processors and three memory modules containing a total of 384k (k = 1024) 36-bit words. This configuration currently is rated at a capacity of about 55 fairly demanding users such that most trivial requests obtain response in one to five seconds. (Future design iterations are expected to increase the capacity rating.) Several times a day during the off-peak usage hours the system is dynamically reconfigured into two systems: a reduced capacity service system and an independent development system. The development system is used for testing those hardware and software changes which cannot be done under normal service operation.

The reliability of the round-the-clock system operation described above has been a matter of great concern, for in any on-line real-time system the impact of mishaps is usually far more severe than in batch processing systems. In an on-line system especially important considerations are:

(1) the time required before the system is usable again following a mishap,
(2) the extra precautions required for restoring possibly lost files, and
(3) the psychological stress of breaking the interactive dialogue with users who were counting on system availability.

Because of the importance of these considerations, careful logs are kept of all Multics "crashes" (i.e., system service disruption for all active users) at M.I.T. in order that analysis can reveal their causes. These analyses indicate currently an average of between one and two crashes per 24 hour day. These crashes have no single cause. Some are due to hardware failures, others to operator error and still others to software bugs introduced during the course of development. At the two other sites where Multics is operated, but where active system development does not take place, there have been almost no system failures traced to software.

TABLE I—A comparison of the system development and use periods of CTSS and Multics. The Multics development period is not significantly longer than that for CTSS despite the development of about 10 times as much code for Multics as for CTSS and a geographically distributed staff. Although reasons for this similarity in time span include the use of a higher-level programming language and a somewhat larger staff, the use of CTSS as a develoΩment tool for Multics was of pivitol importance.

System	*Development Only*	*Development + Use*	*Use Only*
CTSS	1960-1963	1963-1965	1965-present
Multics	1964-1969	1969-present	

Currently the Multics system, including compilers, commands, and subroutine libraries, consists of about 1500 modules, averaging roughly 200 lines of PL/I apiece. These compile to produce some 1,000,000 words of procedure code. Another measure of the system is the size of the resident supervisor which is about 30k words of procedure and, for a 55 user load, about 36k words of data and buffer areas.

Because the system is so large, the most powerful maintenance tool available was chosen—the system itself. With all of the system modules stored on-line, it is easy to manipulate the many components of different versions of the system. Thus it has been possible to maintain steadily for the last year or so a pace of installing 5 or 10 new or modified system modules a day. Some three-quarters of these changes can be installed while the system is in operation. The remainder, pertaining to the central supervisor, are installed in batches once or twice a week. This on-line maintenance capability has proven indispensable to the rapid development and maintenance of Multics since it permits constant upgrading of the user interface without interrupting the service. We are just beginning to see instances of user-written applications which require this same capability so that the application users need not be interrupted while the software they are using is being modified.

The software effort which has been spent on Multics is difficult to estimate. Approximately 150 man-years were applied directly to design and system programming during the "development-only" period of Table I.

Since then we estimate that another 50 man-years have been devoted to improving and extending the system. But the actual cost of a single successful system is misleading, for if one starts afresh to build a similar system, one must compensate for the non-zero probability of failure.

THE APPEARANCE OF MULTICS TO ITS USERS

Having reviewed the background of the project, we may now ask who are the users of the Multics system and what do the facilities that Multics provides mean to these users. Before answering, it is worth describing the generic user as "viewed" by Multics. Although from the system's point of view all users have the same general characteristics and interface with it uniformly, no single human interface represents the Multics machine. That machine is determined by each user's initial procedure coupled with those functions accessible to him. Thus there exists the potential to present each Multics user with a unique external interface.

However, Multics does provide a native internal program environment consisting of a stack-oriented, pure-procedure, collection of PL/I procedures imbedded in a segmented virtual memory containing all procedures and data stored on-line. The extent to which some, all, or none of this internal environment is visible to the various users is an administrative choice.

The implications of these two views—both the external interface and the internal programming environment—are discussed in terms of the following categories of users:

- System programmers and user application programmers responsible for writing system and user software.
- Administrative personnel responsible for the management of system resources and privileges.
- The ultimate users of applications systems.
- Operations and hardware maintenance personnel responsible, respectively, for running the machine room and maintaining the hardware.

Multics as viewed by system and subsystem programmers

The machine presented to both the Multics system programmer and the application system programmer is the one with which we have the most experience; it is the raw material from which one constructs other environments. It is worth reemphasizing that the only differentiation between Multics system programmers and user programmers is embodied in the access control mechanism which determines what on-line information can be referenced; therefore, what are apparently two groups of users can be discussed as one.

Major interfaces presented to programmers on the Multics system can be classified as the program preparation and documentation facilities and the program execution and debugging environment. They will be touched upon briefly, in the order used for program preparation.

Program preparation and documentation

The facilities for program preparation on Multics are typical of those found on other time-sharing systems, with some shifts in emphasis (see the Appendix). For example, programmers consider the file system sufficiently invulnerable to physical loss that it is used casually and routinely to save all information. Thus, the punched card has vanished from the work routine of Multics programmers and access to one's programs and the ability to work on them are provided by the closest terminal.

As another example, the full ASCII character set is employed in preparing programs, data, and documentation, thereby eliminating the need for multiple text editors, several varieties of text formatting and comparison programs, and multiple facilities for printing information both on-line and off-line. This generalization of user interfaces facilitates the learning and subsequent use of the system by reducing the number of conventions which must be mastered.

Finally, because the PL/I compiler is a large set of programs, considerable attention was given to shielding the user from the size of the compiler and to aiding him in mastering the complexities of the language. As in many other time-sharing systems, the compiler is invoked by issuing a simple command line from a terminal exactly as for the less ambitious commands. No knowledge is required of the user regarding the various phases of compilation, temporary files required, and optional capabilities for the specialist; explanatory "sermons" diagnosing syntactic errors are delivered to the terminal to effect a self-teaching session during each compilation. To the programmer, the PL/I compiler is just another command.

Program execution environment

Another set of interfaces is embodied in the implementation environment seen by PL/I programmers. This environment consists of a directly addressable virtual memory containing the entire hierarchy of on-line information, a dynamic linking facility which searches this hierarchy to bind procedure references, a device-independent input/output[16] system,* and program debugging and metering facilities. These facilities enjoy a symbiotic relationship with the PL/I procedure environment used both to implement them and to implement user facilities co-existing with them. Of major significance is that the natural internal environment provided and required by the system is exactly that environment expected by PL/I procedures. For example, PL/I pointer variables, call and return statements, conditions, and static and automatic storage all correspond directly to mechanisms provided in the internal environment. Consequently, the system supports PL/I code as a matter of course.

The main effect of the combination of these features is to permit the implementer to spend his time concentrating on the logic of his problem; for the most part he is freed from the usual mechanical problems of storage management and overlays, input/output device quirks, and machine-dependent features.

Some implementation experience

The Multics team began to be much more productive once the Multics system became useful for software development. A few cases are worth citing to illustrate the effectiveness of the implementation environment. A good example is the current PL/I compiler, which is the third one to be implemented for the project, and which consists of some 250 procedures and about 125k words of object code. Four people implemented this compiler in two years, from start to first general use. The first version of the Multics program debugging system, composed of over 3,000 lines of source code, was usable after one person spent some six months of nights and weekends "bootlegging" its implementation. As a last example, a facility consisting of 50 procedures with a total of nearly 4,000 PL/I statements permitting execution of Honeywell 635 programs under Multics became operational after one person spent eight months learning about the GCOS operating system for the 635, PL/I, and Multics, and then implemented the environment. In each of these examples the implementation was accomplished from remote terminals using PL/I.

Multics users have discovered that it is possible to get their programs running very quickly in this environment. They frequently prepare "rough drafts" of programs, execute them, and then improve their overall design and operating strategy using the results of experience obtained during actual operation. As an example, again drawn from the implementation of Multics, the early designs and implementations of the programs supporting the virtual memory[18] made overoptimistic use of variable-sized storage allocation techniques. The result was a functionally correct but inadequately performing set of programs. Nevertheless, these modules were used as the foundation for subsequent work for many months. When they were finally replaced with modules using simplified fixed-size storage techniques, performance improvements of over an order of magnitude were realized. This technique emphasizes two points: first, it is frequently possible to provide a practical, usable facility containing temporary versions of programs; second, often the insight required to significantly improve the behavior of a program comes only after it is studied in operation. As implied in the earlier discussion of design iteration, our experience has been that structural and strategic changes rather than "polishing" (or recoding in assembly language) produce the most significant performance improvements.

In general, we have noticed a significant "amplifier" or "leverage" effect with the use of an effective on-line environment as a system programming facility. Major implementation projects on the Multics system seldom involve more than a few programmers, thereby easing the management and communications problems usually entailed by complex system implementations. As would be expected, the amplification effect is most apparent with the best project personnel.

* The Michigan Terminal System[17] has a similar device-independent input/output system.

Administration of Multics facilities and resources

The problem of managing the capabilities of a computer utility with geographically dispersed subscribers leads to a requirement of decentralized administration. At the apex of an administrative pyramid resides a system administrator with the ability to register new users, confer resource quotas, and generate periodic bills for services rendered. The system administrator deals with user groups called projects. Each group can in turn designate a project administrator who is delegated the authority to manage a budget of system resources on behalf of the project. The project administrator is then free to deal directly with project members without further intervention from the system administrator, thereby greatly reducing the bottlenecks inherent in a completely centralized administrative structure.

Environment shaping

In addition to having immediate control of such resources as secondary storage, port access, and rate of processor usage, the project administrator is also able to define or shape the environment seen by the members of his project when they log into the system. He does this by defining those procedures that can be accessed by members of his project and by specifying the initial procedure executed by each member of his project when he logs in. This environment shaping facility has led to the notion of a private project subsystem on Multics. It combines the administrative and programming facilities of Multics so that a project administrator and a few project implementers can build, maintain, and evolve environments entirely on their own. Thus, some subsystems bear no internal resemblance to the standard Multics procedure environment.

For example, the Dartmouth BASIC[19] compiler executes in a closed subsystem implemented by an M.I.T. student group for use by undergraduate students. The compiler, its object code, and all support routines execute in a simulation of the native environment provided at Dartmouth. The users of this subsystem need little, if any, knowledge of Multics and are able to behave as if logged into the Dartmouth system proper. Other examples of controlled environment subsystems include one to permit many programs which normally run under the GCOS operating system to also run unmodified in Multics. Finally, an APL[20] subsystem allows the user to behave for the most part as if he were logged into an APL machine. The significance of these subsystems is that their implementers did not need to interact with the system administrator or to modify already existing Multics capabilities. The administrative facilities permit each such subsystem to be offered by its supporters as a private service with its own group of users, each effectively having its own private computer system.

Other Multics users

Finally, we observe that the roles of the application user, the system operators and the hardware maintainers as seen by the system are simply those of ordinary Multics users with specialized access to the on-line procedures and data. The effect of this uniformity of treatment is to reduce greatly the maintenance burden of the system control software. One example, of great practical importance, has been the ease with which system performance measurement tools have been prepared for use by the operating staff.

INSIGHTS

So far, we have discussed the status and appearance of the Multics system. A further question is what has been learned in the construction of Multics which is of use to the designers of other systems. Having a bright idea which clearly solves a problem is not sufficient cause to claim a contribution if the idea is to be part of a complex system. In order to establish the real feasibility of an idea, all of its implications and consequences must be followed out. Much of the work on Multics since 1965 has involved following out implications and consequences of the many ideas then proposed for the prototype computer utility. That following out is an essential part of proof of ideas is attested by the difficulties which have been encountered in other engineering efforts such as the development of nuclear fusion power plants and the electric automobile. Not all proposals work out; for example, extended attempts to engineer an atomic powered airplane suggest infeasibility.

Perhaps Multics' most significant single contribution to the state of the art of computer system construction is the demonstration of a large set of fully implemented ideas in a working system. Further, most of these ideas have been integrated without straining the overall design; most additional proposals would not topple the structure. Ideas such as virtual memory access to on-line storage, parallel process organization, routine but controlled information sharing, dynamic linking of procedures, and high-level language implementa-

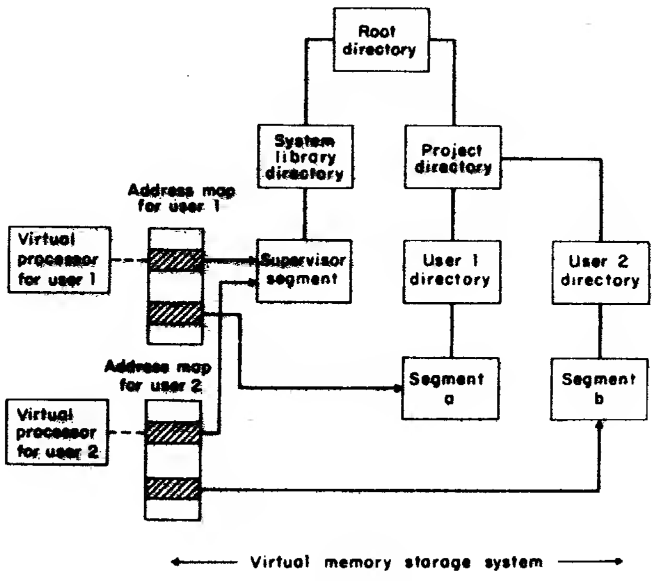


Figure 1—The entire storage hierarchy may be mapped into individual user process address spaces (see arrows) as if contained in primary memory. Illustrated are the sharing of a supervisor segment by user 1 and user 2 and private access to segment a and segment b. The necessary primary storage is simulated by a demand paging technique which moves information between the real primary memory and secondary storage

tion have proven remarkably compatible and complementary.

To illustrate some of the areas of progress in understanding of system organization and construction which have been achieved in Multics, we consider here the following five topics:

1. Modular division of responsibility
2. Dynamic reconfiguration
3. Automatically managed multilevel memory
4. Protection of programs and data
5. System programming language

Modular division of responsibility

Early in the design of Multics a decision had to be made whether or not to treat the segmented virtual memory as a separately usable "feature," independent of a traditionally organized read/write type file system. The alternative, to use the segmented virtual memory as the file system itself, providing the illusion of direct "in-core" access to all on-line storage, was certainly the less conservative approach (see Figure 1). The second approach, which was the one chosen, led to a strong test of the ability of a computing system to support an apparent one-level memory for an arbitrarily large information base. It is interesting that the resulting almost total decoupling between physical storage allocation and data movement on the one hand and directory structure, naming, and file organization on the other led to a remarkably simple and functionally modular structure for that part of the system[18] (see Figure 2).

Another area of Multics in which a high degree of functional modularity was achieved was in the area of scheduling, multiprogramming, and processor management. Because harnessing of multiple processors was an objective from the beginning, a careful and methodical approach to multiplexing processors, handling interrupts, and providing interprocess synchronizing primitives was developed. The resulting design, known as the Multics traffic controller, absorbed into a single, simple module a set of responsibilities often diffused among a scheduling algorithm, the input/output controlling system, the on-line file management system, and special purpose inter-user communication mechanisms.[21]

Finally, with processor management and on-line storage management uncoupled into well-isolated modules, the Multics input/output system was left with the similarly isolatable function of managing streams of data flowing from and to source and sink type devices.[16] Thus, this section of the system concentrates only on switching of the streams, allocation of data buffering areas, and device control strategies.

Each of the divisions of labor described above represents an interesting result primarily because it is so difficult to discover appropriate divisions of complex systems.* Establishing that a certain proposed division results in simplicity, creates an uncluttered interface, and does not interfere with performance, is generally cause for a minor celebration.

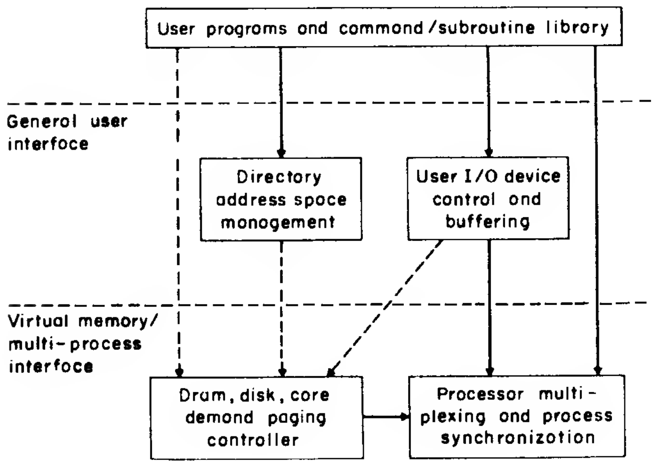

User programs and command/subroutine library
General user interface
Directory address space management
User I/O device control and buffering
Virtual memory/ multi-process interface
Drum, disk, core demand paging controller
Processor multiplexing and process synchronization

Figure 2—Major lines of modular division in Multics. Solid lines indicate calls for services. Dotted lines indicate implicit use of the virtual memory

Dynamic reconfiguration

If the computer utility is ever to become as much a reality as the electric power utility or the telephone communication service, its continued operation must not be dependent upon any single physical component, since individual components will eventually require maintenance. This observation leads an electric power utility to provide procedures whereby an idle generator may be dynamically added to the utility's generating capacity, while another is removed for maintenance, all without any disruption of service to customers. A similar scenario has long been proposed for multiprocessor, multimemory computer systems, in which one would dynamically switch processsors and memory boxes in and out of the operating configuration as needed. Unfortunately, though there have been demonstrated a few "special purpose" designs,* it has not been apparent how to provide for such operations in a general purpose system. A recent thesis[24] proposed a general model for the dynamic binding and unbinding of computation and memory structures to and from ongoing computa-

* See Dijkstra[22] for a further discussion of this point.

* An outstanding example is the American Airlines SABRE system.[23]

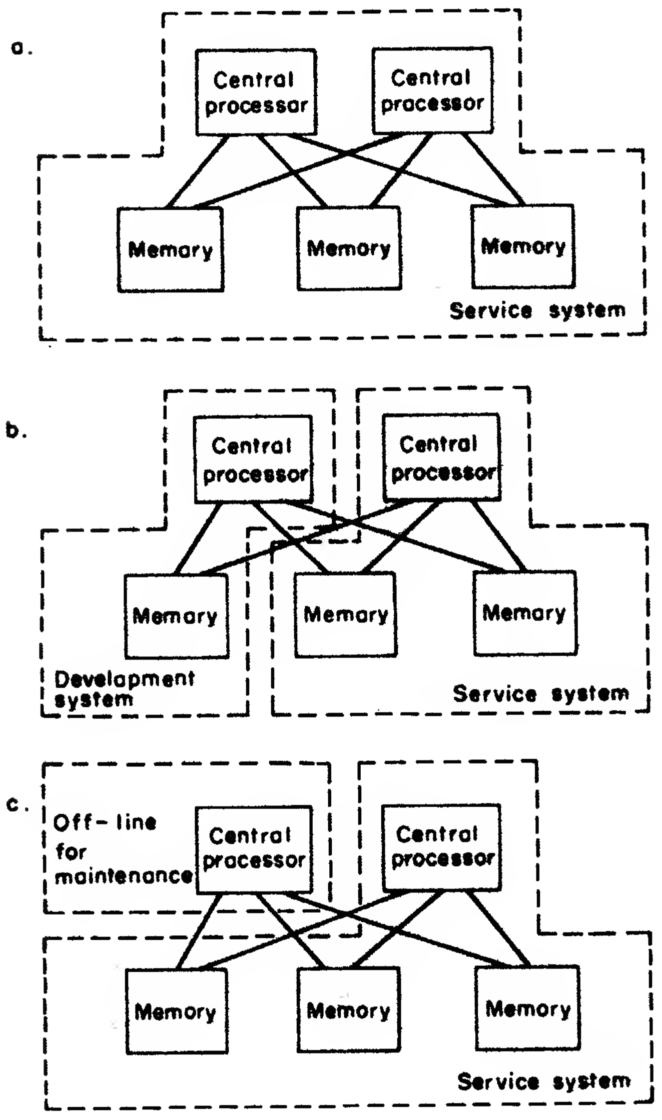


Figure 3—Dynamic reconfiguration permits switching among the three typical operating configurations shown here, without currently logged-in users being aware that a change has taken place

tions. Using this model as a basis, the thesis also proposed a specific implementation for a typical multiprocessor, multimemory computing system. One of the results of this work was the addition to the operating Multics system of the capability of dynamically adding and removing central processors and memory modules as in Figure 3. The usefulness of the idea may be gauged by observing that at M.I.T. five to ten such reconfigurations are performed in a typical 24-hour operating day. Most of the reconfigurations are used to provide a secondary system for Multics development.

Automatically managed multilevel memory

By now it has become accepted lore in the computer system field that the use of automatic management algorithms for memory systems constructed of several levels with different access times can provide a significant reduction of user programming effort. Examples of such automatic management strategies include the buffer memories of the IBM system 370 models 155, 165, and 195[25] and the demand paging virtual memories of Multics, IBM's CP-67[26] and the Michigan Terminal System.[17] Unfortunately, behind the mask of acceptance hides a worrisome lack of knowledge about how to engineer a multilevel memory system with appropriate strategy algorithms which are matched to the load and hardware characteristics. One of the goals of the Multics project has been to instrument and experiment with the multilevel memory system of Multics, in order to learn better how to predict in advance the performance of proposed new automatically managed multilevel memory systems. Several specific aspects of this goal have been explored:

- A strategy to treat core memory, drum, and disk as a three-level system has been proposed, including a "least-recently-used" algorithm for moving information from drum to disk. Such an algorithm has been used for some time to determine which pages should be removed from core memory.[27] The dynamics of interaction among two such algorithms operating at different levels are weakly understood, and some experimental work should provide much insight. The proposed strategy will be implemented, and then compared with the simpler present strategy which never moves things from drum to disk, but instead makes educated "guesses" as to which device is most appropriate for the permanent residence of a given page. If the automatic algorithm is at least as good as the older, static one, it would represent an improvement in overall design by itself, since it would automatically track changes in user behavior, while the static algorithm requires attention to the validity of its guesses.
- A scheme to permit experimentation with predictive paging algorithms was devised. The scheme provides for each process a list of pages to be preloaded whenever the process is run, and a second list to be immediately purged whenever the process stops. The updating of these lists is controlled by a decision table exercised every time the process stops running. Since every page of the Multics virtual memory is potentially shared, the decision table represents a set of heuristics designed to separate out those which are probably not being shared at the moment.

- A series of measurements was made to establish the effectiveness of a small hardware associative memory used to hold recently accessed page descriptors. These measurements established a profile of hit ratio (probability of finding a page descriptor in the associative memory) versus associative memory size which should be useful to the designers of virtual memory systems.[28]
- A set of models, both analytic and simulation, was constructed to try to understand program behavior in a virtual memory. So far, two results have been obtained. One is the finding that a single program characteristic (the mean execution time before encountering a "missing" page in the virtual memory as a function of memory size) suffices to provide a quite accurate prediction of paging and idle overheads. The second is direct calculation of the distribution of response times under multiprogramming. Having available the entire response time distribution, rather than just averages, permits estimation of the variance and 90-percentile points of the distribution, which may be more meaningful than just the average. A doctoral thesis is in progress on this topic.

Although the immediate effect of each of these investigations is to improve the understanding or performance of the current version of Multics, the long-range payoff in methodical engineering using better-understood memory structures is also evident.

Protection of programs and data

A long-standing objective of the public computer utility has been to provide facilities for the protection of executing programs from one another, so that users may with confidence place appropriate control on the release of their private information. In 1967, a mechanism was proposed[29] and implemented in software which generalized the usual supervisor-user protection relationship. This mechanism, named "rings of protection," provides user-written subsystems with the same protection from other users that the supervisor has, yet does not require that the user-written subsystem be incorporated into the supervisor. Recently, this approach was brought under intense review, with two results:

- A hardware architecture which implements the mechanism was proposed.[30] One of the chief features of the proposed architecture is that subroutine calls from one protection ring to another use exactly the same mechanisms as do subroutine calls among procedures within a protection area. The proposal appears sufficiently promising that it is included in the specifications for the next genera tion of hardware to be used for Multics.
- As an experiment in the feasibility of a multi-layered supervisor, several supervisor procedures which required protection, but not all supervisor privileges, were moved into a ring of protection intermediate between the users and the main supervisor. The success of this experiment established that such layering is a practical way to reduce the quantity of supervisor code which must be given all privileges.

Both of these results are viewed as steps toward first, a more complete exploitation and understanding of rings of protection, and later, a less constrained organization of the type suggested by Evans and LeClerc[31] and by Lampson.[32] But more importantly, rings of protection appear applicable to any computer system using a segmented virtual memory. Two doctoral theses are under way in this area.

System programming language

Another technique of system engineering methodology being explored within the Multics project is that of higher level programming language for system implementation. The initial step in this direction (which proved to be a very big step) was the choice of the PL/I language for the implementation of Multics. By now, Multics offers an extensive case study in the viability of this strategy. Not only has the cost of using a higher level language been acceptable, but increased maintainability of the software has permitted more rapid evolution of the system in response to development ideas as well as user needs. Three specific aspects of this experience have now been completed:

- The transition from an early PL/I subset compiler[14] to a newer compiler which handles almost the entire language was completed. This transition was carried out with performance improvement in practically every module converted in spite of the larger language involved. The significance of the transition is the demonstration that it is not necessary to narrow one's sights to a "simple" subset language for system programming. If the language is thoroughly understood, even a language as complex as the full PL/I can be effectively used. As a result, the same language and compiler provided for users can also be used for system implementation, thereby minimizing maintenance, confusion, and specialization.
- Notwithstanding the observation just made, the time required to implement a full PL/I compiler is still too great for many situations in which the

compiler implementation cannot be started far enough in advance of system coding. For this reason, there is considerable interest in defining a smaller language which is easily compilable, yet retains the features most important for system implementation. On the basis of the experience of programming Multics in a subset of PL/I, such a language was defined but not implemented, since it was not needed.[33]

- A census of Multics system modules reveals how much of the system was actually coded in PL/I, and reasons for use of other languages. Roughly, of the 1500 system modules, about 250 were written in machine language. Most of the machine language modules represent data bases or small subroutines which execute a single privileged instruction. (No attempt was made to provide either a data base compiler or PL/I built-in functions for specialized hardware needs.) Significantly, only a half dozen areas (primarily in the traffic controller, the central page fault path, and interrupt handlers) which were originally written in PL/I have been recoded in machine language for reasons of squeezing out the utmost in performance. Several programs, originally in machine language, have been recoded in PL/I to increase their maintainability.

As with the earlier topics, the implications of this work with PL/I should be felt far beyond the Multics system. Most implementers, when faced with the economic uncertainties of a higher-level language, have chosen machine language for their central operating systems. The experience of PL/I in Multics when added to the expanding collection of experience elsewhere[34] should help reduce the uncertainty.

In a research project as large, long, and complex as Multics, any paper such as this must necessarily omit many equally significant ideas, and touch only a few which may happen to have wide current interest. It is the purpose of individual and detailed technical papers to explain these and other ideas more fully. The bibliography found in Reference 35 contains over twenty such technical papers.

Immediate future plans

The Multics software is continuing to evolve in response to user needs and improved understanding of its organization. In 1972 a new hardware base for Multics will be installed by the Information Processing Center at M.I.T. for use by the M.I.T. computing community. This program compatible hardware base contains small but significant architectural extensions to the current hardware. The circuit technology used will be that of the Honeywell 6080 computer. The substantial changes include:

(1) replacement of the high-performance paging drum initially with bulk core and, when available, LSI memory, and
(2) implementation of rings of protection as part of the paging and segmentation hardware.

Wherever possible the strategy of using off-the-shelf standard equipment rather than specially engineered units for Multics has been followed. This strategy is intended to simplify maintenance.

CONCLUSIONS

There are many conclusions which could possibly be drawn from the experience of the Multics project. Of these, we consider four to be major and worthy of note. First, we feel it is clear that it is possible to achieve the goals of a prototype computer utility. The current implementation of Multics provides a measure of the mechanisms required. Moreover, the specific implementation of the system, because it has been written in PL/I, forms a model fo- other system designers to draw upon when constructing similar systems.

Second, the question of whether or not the specific software features and mechanisms which were postulated for effective computer utility operation are desirable has now been tested with specific user experience. Although the specific mechanisms implemented subsequently may be superseded by better ones, it is certainly clear that the improvement of the user environment which was wanted has been achieved.

Third, systems of the computer utility class must evolve indefinitely since the cost of starting over is usually prohibitive and the many-year lead time required may be equally unacceptable. The requirement of evolvability places stringent demands on design, maintainability, and implementation techniques.

Fourth and finally, the very act of creating a system which solves many of the problems posed in 1965 has opened up many new directions of research and development. It would appear almost a certainty that increased user aspirations will continue to require intensive work in the areas of computer system principles and techniques.

In closing, perhaps we should take note that in the seven years since Multics was proposed, a great many other systems have also been proposed and constructed;

many of these have developed similar ideas.* In most cases, their designers have developed effective implementations which are directed to a different interpretation of the goals, or to a smaller set of goals than those required for the complete computer utility. This diversity is valuable, and probably necessary, to accomplish a thorough exploration of many individually complex ideas, and thereby to meet a future which holds increasing demand for systems which embrace the totality of computer utility requirements.

ACKNOWLEDGMENT

It is impossible to acknowledge accurately the contributions of all the individuals or even the several organizations which have given various forms of support to the development of Multics over the past seven years. As would be expected of any multi-organization project spanning several years there has been a turnover in the personnel involved. As the individual contributors now number in the hundreds, proper recognition eannot be given here. Instead, since the development of significant features and designs of Multics has occurred in specifie areas and disciplines such as input/output, virtual memory design, languages, and resource multiplexing, a more accurate delineation of achievements should be made in specialized papers. So in the end we must defer to the authors of individual papers, past and future, to acknowledge the efforts of some of the many contributors who have made the evolution of Multics possible.

REFERENCES

1 F J CORBATÓ M M DAGGETT R C DALEY
An experimental time-sharing system
AFIPS Conf Proc 21 Spartan Books 1962 pp 335-344

2 P A CRISMAN Ed
The compatible time-sharing system: A programmer's guide
2nd ed MIT Press Cambridge Massachusetts 1965

3 F J CORBATÓ V A VYSSOTSKY
Introduction and overview of the Multics system
AFIPS Conf Proc 27 1965 FJCC Spartan Books Washington D C 1965 pp 185-196

4 E L GLASER et al
System design of a computer for time-sharing application
AFIPS Conf Proc 27 1965 FJCC Spartan Books Washington D C 1965 pp 197-202

5 V A VYSSOTSKY et al
Structure of the Multics supervisor
AFIPS Conf Proc 27 1965 FJCC Spartan Books Washington D C 1965 pp 203-212

6 R C DALEY P G NEUMANN
A general-purpose file system for secondary storage
AFIPS Conf Proc 27 1965 FJCC Spartan Books Washington D C 1965 pp 213-229

7 J F OSSANNA et al
Communication and input/output switching in a multiplex computing system
AFIPS Conf Proc 27 1965 FJCC Spartan Books Washington D C 1965 pp 231-241

8 E E DAVID JR R M FANO
Some thoughts about the social implications of accessible computing
AFIPS Conf Proc 27 1965 FJCC Spartan Books Washington D C 1965 pp 243-247

9 E I ORGANICK
The Mutlics system: An examination of its structure
MIT Press (in press) Cambridge Massachusetts and London England

10 R W WATSON
Timesharing system design concepts
McGraw-Hill Book Company New York 1970

11 W T COMFORT
A computing system design for user service
AFIPS Conf Proc 27 1965 FJCC Spartan Books Washington D C 1965 pp 619-626

12 A S LETT W L KONIGSFORD
TSS/360: A time-shared operating system
AFIPS Conf Proc 33 1968 FJCC Thompson Books pp 15-28

13 R E SCHWEMM
Experience gained in the development and use of TSS/360
AFIPS Conf Proc 40 1972 SJCC AFIPS Press (This volume)

14 F J CORBATÓ
PL/I as a tool for system programming
Datamation 15 6 May 1969 pp 68-76

15 R A FREIBURGHOUSE
The Multics PL/I compiler
AFIPS Conf Proc 35 1969 FJCC AFIPS Press 1969 pp 187-199

16 R J FEIERTAG E I ORGANICK
The Multics input-output system
ACM Third Symposium on Operating Systems Principles October 18-20 1971 pp 35-41

17 M T ALEXANDER
Organization and features of the Michigan terminal system
AFIPS Conf Proc 40 1972 SJCC AFIPS Press (This volume)

18 A BENSOUSSAN C T CLINGEN R C DALEY
The Multics virtual memory
ACM Second Symposium on Operating System Principles October 20-22 1969 Princeton University pp 30-42

19 *BASIC*
Fifth Edition Kiewit Computation Center Dartmouth College September 1970

* Some examples which have not already been mentioned include: the TENEX system of Bolt, Beranek and Newman, the VENUS system of Mitre Corp., the MU5 at Manchester University, RC-4000 of Regnecentralen, 5020 TSS of Hitachi Corp., DIPS-1 of Nippon Telephone, the Japanese National Computer Project, the PDP-10/50 TSS of Digital Equipment Corp., the BCC-500 of Berkeley Computer Corp., I.T.S. of the M.I.T. Artificial Intelligence Laboratory, Exec-8 of Univac, System 3 and 7 and the SPECTRA 70/46 of RCA, Star-100 of CDC, UTS of Xerox Data Systems, the 6700 system of Burroughs, and the Dartmouth Time-Sharing System.

20 *APL/360 user's manual*
IBM form number GH20-0683-1 March 1970

21 J H SALTZER
Traffic control in a multiplexed computer system
ScD Thesis MIT Department of Electrical Engineering 1966 Also available as Project MAC technical report TR-30

22 E W DIJKSTRA
The structure of the 'THE'-Multiprogramming system
Comm ACM 11 5 May 1968 pp 341-346

23 R W PARKER
The Sabre system
Datamation 11 9 September 1965 pp 49-52

24 R R SCHELL
Dynamic reconfiguration in a modular computer system
PhD Thesis MIT Department of Electrical Engineering 1971 Also available as Project MAC technical report TR-86

25 C J CONTI
Concepts for buffer storage
IEEE Computer Group News March 1969 pp 9-13

26 R A MEYER L H SEAWRIGHT
A virtual machine time-sharing system
IBM Systems Journal 9 3 1970 pp 199-218

27 F J CORBATÓ
A paging experiment with the Multics system
In Honor of P M Morse MIT Press Cambridge Massachusetts 1969 pp 217-228

28 M D SCHROEDER
Performance of the GE-645 associative memory while Multics is in operation
ACM Workshop on System Performance Evaluation April 1971 pp 227-245

29 R M GRAHAM
Protection in an information processing utility
Comm ACM 11 5 May 1968 pp 365-369

30 M D SCHROEDER J H SALTZER
A hardware architecture for implementing protection rings
ACM Third Symposium on Operating Systems Principles October 18-20 1971 pp 42-54

31 D C EVANS J Y LeCLERC
Address mapping and the control of access in an interactive computer
AFIPS Conf Proc 30 1967 SJCC Thompson Books 1967 pp 23-30

32 B W LAMPSON
An overview of the CAL time-sharing system
Computer Center University of California Berkeley September 5 1969

33 D D CLARK R M GRAHAM J H SALTZER M D SCHROEDER
Classroom information and computing service
MIT Project MAC Technical Report TR-80 January 11 1971

34 J E SAMMET
Brief survey of languages used for systems implementation
SIGPLAN Notices 6 9 October 1971 pp 1-19

35 *The multiplexed information and computing service: Programmers' manual*
MIT Project MAC Rev 10 1972 (Available from the MIT Information Processing Center)

APPENDIX: A CHECKLIST OF MULTICS FEATURES

Following is a checklist of currently available features and facilities of Multics. Although many of the features are described in cryptic and untranslated local jargon, one can at least obtain a feel for the range of facilities now provided. Further information on most of these features may be found in the Multics Programmers' Manual.[35]

Interactive Time-Sharing Facilities
- file editors
- file manipulation (rename/move/delete)
- personal command abbreviations
- recursive command language
- source language debugging with breakpoints
- subroutine call tracer
- can stop any running command or program

Programming Languages
- PL/I
- FORTRAN
- BASIC*
- APL
- LISP
- BCPL
- ALM (assembly language/Multics)

Information Storage System
- configuration independent
- accessed through virtual memory (segments)
- access control lists by user and project
- links to segments of other users
- hierarchical directory (catalog) arrangement
- public library facilities
- sharing at all levels
- multiple segment names (synonyms)
- separate control of read, write, and execute

Programming Environment
- segmented virtual memory
- dynamic linking of procedures and data, or prelinking
- interprocess communication
- independent of configuration
- uniform error handling mechanism
- user definable protection rings
- microsecond calendar clock with interrupt
- program interrupt signal from console

Input and Output
- standard typewriter interface for device independence
- ASCII character set used throughout
- input characters converted to canonical form
- erase and kill editing on typed input

I/O streams switchable during execution
magnetic tape, printer, card punch, card reader
typewriter terminals: IBM 2741, 1050
Teletype 37, 33, 35
Dura, Datel, Execuport, Terminet-300
graphic support library (devices: ARDS, IMLAC, DEC 338)
ARPA network
interfaces at three levels:
- formatted data conversion
- bit stream control
- full device control

Management Facilities
- passwords required for login
- project may interpose authentication procedure
- decentralized projects
- accounting, billing, and quotas
- on-line probing and account adjustment
- operator or system initiated logout of users
- unlisted and anonymous users
- limited service system
- dynamic reconfiguration of memories and processors
- system performance metering for parameter adjustment
- project-imposed starting procedure

Communication Facilities
- interuser mail
- help command; help files
- message of the day
- on-line error reporting and consultation service
- on-line user graffiti board
- operations message broadcast to logged-in users

Absentee Facilities
- priority/defer queues for printer, card punch
- queued translator facility
- general absentee job facility

Reliability Measures
- weekly file copies onto tape
- daily disk/drum copy onto tape
- incremental file copies onto tape, ½ hour behind use
- salvager to clean up files after system crash
- emergency shutdown entry to system

Maintenance Features
- on-line library change, no disruption of current users
- entire system source on-line, maintenance tools
- system checkout on small hardware configuration
- on-line performance monitoring of
  - multiprogramming
  - paging traffic
  - drum/disk usage
  - typewriter traffic
- user performance feedback:
  - cpu time and paging load on each command
  - page trace always operating
  - subroutine call counters

Private Project Subsystems
- project providable command interface
- Dartmouth environment*
- student environment

Miscellaneous Facilities
- desk calculators
- sort command
- memorandum formatting and typing subsystem
- user-provided list of programs to be automatically executed when user logs in
- GCOS environment

* The BASIC system and the Dartmouth environment were developed at Dartmouth College. They are used at M.I.T. by permission of Dartmouth College.

The Multics Virtual Memory: Concepts and Design

by A. Bensoussan, C.T. Clingen, and R.C. Daley. Reprinted from Communications of the ACM 15, 5, May, 1972, pp. 308-318, with permission. Copyright 1972 by the Association for Computing Machinery.

After four sections of relatively elementary introduction, this paper delves deeply into the mechanisms required to support a virtual memory system in which all on-line storage is addressed directly by the processor. This virtual memory system is probably the most important conceptual departure introduced by Multics. It is of special interest to writers of complex application subsystems which manipulate data bases shared by several users. The power of the Multics virtual memory as a tool to reduce programming effort is illustrated in MPM Introduction Chapter Four.

Since this paper is a recent one, the terminology is quite up-to-date, although the description given here is abstracted somewhat from the actual implementation to avoid cluttering details. Large copies of figures four and five, which did not reproduce well in the original publication, will be found after the last page of the paper.

2-18

The Multics Virtual Memory: Concepts and Design

A. Bensoussan, C.T. Clingen
Honeywell Information Systems, Inc.*
and
R.C. Daley
Massachusetts Institute of Technology†

As experience with use of on-line operating systems has grown, the need to share information among system users has become increasingly apparent. Many contemporary systems permit some degree of sharing. Usually, sharing is accomplished by allowing several users to share data via input and output of information stored in files kept in secondary storage. Through the use of segmentation, however, Multics provides direct hardware addressing by user and system programs of all information, independent of its physical storage location. Information is stored in segments each of which is potentially sharable and carries its own independent attributes of size and access privilege.

Here, the design and implementation considerations of segmentation and sharing in Multics are first discussed under the assumption that all information resides in a large, segmented main memory. Since the size of main memory on contemporary systems is rather limited, it is then shown how the Multics software achieves the effect of a large segmented main memory through the use of the Honeywell 645 segmentation and paging hardware.

Key Words and Phrases: operating system, Multics, virtual memory, segmentation, information sharing, paging, memory management, memory hierarchy

CR Categories: 4.30, 4.31, 4.32

1. Introduction

In the past few years several well-known systems have implemented large virtual memories which permit the execution of programs exceeding the size of available core memory. These implementations have been achieved by demand paging in the Atlas computer [11], allowing a program to be divided physically into pages only some of which need reside in core storage at any one time, by segmentation in the B5000 computer [15], allowing a program to be divided logically into segments, only some of which need be in core, and by a combination of both segmentation and paging in the Honeywell 645 [3, 12] and the IBM 360/67 [2] for which only a few pages of a few segments need be available in core while a program is running.

As experience has been gained with remote-access, multiprogrammed systems, however, it has become apparent that, in addition to being able to take advantage of the direct addressibility of large amounts of information made possible by large virtual memories, many applications also require the rapid but controlled sharing of information stored on-line at the central facility. In Multics (*Multi*plexed *I*nformation and *C*omputing *S*ervice) segmentation provides a generalized basis for the direct accessing and sharing of on-line information by satisfying two design goals: (1) it must be possible for all on-line information stored in

Minor revision of a paper presented at an ACM Symposium on Operating System Principles, Princeton University, October 20–22, 1969. Work reported herein was supported (in part) by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(1). *Honeywell Information Systems, Inc., Cambridge, MA 02142. †Cambridge, MA 02142.

the system to be addressed directly by a processor and hence referenced directly by any computation; (2) it must be possible to control access, at each reference, to all on-line information in the system.

The fundamental advantage of direct addressibility is that information copying is no longer mandatory. Since all instructions and data items in the system are processor-addressible, duplication of procedures and data is unnecessary. This means, for example, that core images of programs need not be prepared by loading and binding together copies of procedures before execution; instead, the original procedures may be used directly in a computation. Also, partial copies of data files need not be read, via requests to an I/O system, into core buffers for subsequent use and then returned, by means of another I/O request, to their original locations; instead the central processor executing a computation can directly address just those required data items in the original version of the file. This kind of access to information promises a very attractive reduction in program complexity for the programmer.

If all on-line information in the system may be addressed directly by any computation, it becomes imperative to be able to limit or control access to this information both for the self-protection of a computation from its own mishaps, and for the mutual protection of computations using the same system hardware facilities. Thus it becomes desirable to compartmentalize or package all information in a directly-addressible memory and to attach access attributes to these information packages describing the fashion in which each user may reference the contained data and procedures. Since all such information is processor-addressible, the access attributes of the referencing user must be enforced upon each processor reference to any information package.

Given the ability to directly address all on-line information in the system, thereby eliminating the need for copying data and procedures, and given the ability to control access to this information, controlled sharing among several computations then follows as a natural consequence.

In Multics, segments are packages of information which are directly addressed and which are accessed in a controlled fashion. Associated with each segment is a set of access attributes for each user who may access the segment. These attributes are checked by hardware upon each segment reference by any user. Furthermore, *all* on-line information in a Multics installation can be directly referenced as segments while in other systems most on-line information is referenced as files.

This paper discusses the properties of an "idealized" Multics memory comprised entirely of segments referenced by symbolic name, and describes the simulation of this idealized memory through the use of both specialized hardware and system software. The result of this simulation is referred to as the Multics virtual memory. Although the Multics virtual memory has been discussed elsewhere [3, 6, 7] at the conceptual level or in its earlier forms, the implementation presented here represents a mechanism resulting from several consecutive implementations leading to an effective realization of the design goals.

2. Segmentation

A basic motivation behind segmentation is the desire to permit information sharing in a more automatic and general manner than provided by nonsegmented systems. Sharing must be accomplished without duplication of information and access to the shared information must be controlled not only in secondary memory but also in main memory.

In most existing systems that provide for information sharing, the two requirements mentioned above are not met. For example, in the CTSS system [5], information to be shared is contained in files. In order for several users to access the information recorded in a file, a *copy* of the desired information is placed in a buffer in each user's core image. This requires an explicit, programmer-controlled I/O request to the file system, at which time the file system checks whether the user has appropriate access to the file. During execution, the user program manipulates this copy and not the file. Any modification or updating is done on the copy and can be reflected in the original file only by an explicit I/O request to the file system, at which time the file system determines whether the user has the right to change the file.

In nonsegmented systems, the use of core images makes it nearly impossible to control access to shared information in core. Each program in execution is assigned a logically contiguous, bounded portion of core memory or paged virtual memory. Even if the nontrivial problem of addressing the shared information in core were solved, access to this information could not be controlled without additional hardware assistance. Each core image consists of a succession of anonymous words that cannot be decomposed into the original elementary parts from which the core image was synthetized. These different parts are indistinguishable in the core image; they have lost their identity and thereby have lost all their attributes, such as length, access rights, and name. As a consequence, nonsegmented hardware is inadequate for controlled sharing in core memory. Although attempts to share information in core memory have been made with nonsegmented hardware, they have resulted in each instance being a special case which must be preplanned at the supervisory level. For example, if all users are to share a compiler in main memory, it is imperative that none of them be able to alter the part of main memory where the compiler resides. The hardware "privileged" mode used by the supervisor is often the only means of protecting shared information in main memory. In order to protect the shared compiler, it is made accessible only in this privileged mode. The compiler can no longer be regarded as a user procedure; it has to be accessed through a supervisor call like any other part of the supervisor, and must be coded to respect any conventions which may have been established for the supervisor.

In segmented systems, hardware segmentation can be used to divide a core image into several parts, or segments [10]. Each segment is accessed by the hardware through a segment descriptor containing the segment's attributes. Among these attributes are access rights that the hardware interprets on each program reference to the segment for a specific user. The absolute core location of the beginning of a segment and its length are also attributes interpreted by the hardware at each reference, allowing the segment to be relocated anywhere in core and to grow and shrink independently of other segments. As a result of hardware checking of access rights, protection of a shared compiler, for example, becomes trivial since the compiler can reside in a segment with only the "execute" attribute, thus permitting users to execute the compiler but not to change it.

In most segmented systems, a user program must first call the supervisor to associate a segment descriptor with a specific file before the program can directly access the information in the file. If the number of files the user program must reference exceeds the number of segment descriptors available to the user, the user program is forced to call the supervisor again to free segment descriptors currently in use so that they can be reused to access other information. Furthermore, if the number of segment descriptors is insufficient to provide simultaneous direct access to each distinct file required by this program, the user must then provide for some means of buffering this information. Buffering, of course, requires that information from more than one file be copied and coalesced with other distinctly different information having potentially different attributes. Once the information is copied and merged, the identity of the original information is lost, thus making it impossible for the information to be shared with other user programs. In addition, this form of user-controlled segment descriptor allocation and buffering of information requires a significant amount of preplanning by the user.

In Multics, the number of segment descriptors available to each computation is sufficiently large to provide a segment descriptor for each file that the user program needs to reference in most applications. The availability of a large number of segment descriptors to each computation makes it practical for the Multics supervisor to associate segment descriptors with files upon first reference to the information by a user program, relieving the user from the responsibility of allocating and deallocating segment descriptors. In addition, the relatively large number of segment descriptors eliminates the need for buffering, allowing the user program to operate directly on the original information rather than on a copy of the information. In this way, all information retains its identity and independent attributes of length and access privilege regardless of its physical location in main memory or on secondary storage. As a result, the Multics user no longer uses files; instead he references all information as segments, which are directly accessible to his programs.

To Multics users, all memory appears to be composed of a large number of independent linear core memories, each associated with a descriptor. A user program can create a segment by issuing a call to the supervisor, giving, as arguments, the appropriate attributes such as symbolic segment name, name of each user allowed to access the segment with his respective access rights, etc. The supervisor then finds an unused descriptor where it stores the segment attributes. The segment having been created, the user program can now address any word of the corresponding linear memory by the pair (name, *i*) where "name" is the symbolic name of the segment and "*i*" is the word number in the linear memory. Furthermore, any other user can reference word number *i* of this segment also by the pair (name, *i*) but he can access it only according to the access rights he was given by the creator and which are recorded in the descriptor. Combinations of the "read," "write," "execute" and "append" access rights [6] are available in Multics.

A simple representation of this memory, referred to as the Multics idealized memory, is shown in Figure 1.

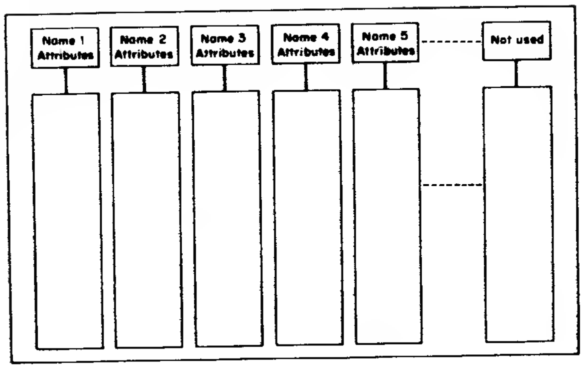
3. Paging

In a system in which the maximum size of any segment was very small compared to the size of the entire core memory, the "swapping" of complete segments into and out of core would be feasible. Even in such a system, if all segments did not have the same maximum size, or had the same maximum size but were allowed to grow from initially smaller sizes, there remains the difficult core management problem of providing space for segments of different sizes. Multics, however, provides for segments of sufficient maximum size so that only a few can be entirely core-resident at any one time. Also, these segments can grow from any initial size smaller than the maximum permissible size.

By breaking segments into equal-size parts called *pages* and providing for the transportation of individual pages to and from core as demand dictates, the disadvantages of fragmentation are incurred, as explained by Denning [9]. However, several practical problems encountered in the implementation of a segmented virtual memory are solved.

First, since pages are all of equal size, space allocation is immensely simplified. The problems of "com-

Fig. 1. Multics idealized memory.



pacting" information in core and on secondary storage, characteristic of systems dealing with variable-sized segments or pages, are thereby eliminated.

Second, since only the referenced page of a segment need be in core at any one instant, segments need not be small compared to core memory.

Third, "demand paging" permits advantage to be taken of any locality of reference peculiar to a program by transporting to core only those pages of segments which are currently needed. Any additional overhead associated with demand paging should of course be weighed against the alternative inefficiencies associated with dedicating core to entire segments which must be swapped into core but which may be only partly referenced.

Finally, demand paging allows the user a greater degree of machine independence in that a large program designed to run well in a large core memory configuration will continue to run at reduced performance on smaller configurations.

4. The Multics Virtual Memory

Multics simulates the idealized memory, represented in Figure 1, using the segmentation and paging features of the 645 assisted by the appropriate software features. The result of the simulation is referred to as the "Multics Virtual Memory." The user can keep a large number of segments in this memory and reference them by symbolic name; upon first reference to a segment, the supervisor automatically transforms the symbolic name into the appropriate hardware address which is directly used by the processor for subsequent references.

The remainder of this paper explains the addressing mechanism in the 645 and describes how the Multics supervisor simulates the Multics idealized memory.

5. The Honeywell 645 Processor

The features of the 645 processor which are of interest for the implementation of the Multics virtual memory are segmentation and paging.

5.1 Segmentation

Any address in the 645 processor consists of a pair of integers $[s, i]$. "s" is called the *segment number*; "i" the index within the segment. The range of "s" and "i" is 0 to $2^{18} - 1$. Word $[s, i]$ is accessed through a hardware register which is the sth word in a table called a *descriptor segment* (DS). The descriptor segment is in core memory and its absolute address is recorded in a processor register called a *descriptor base register* (DBR). Each word of the DS is called a *segment descriptor word* (SDW); the sth SDW will be referred to as SDW(s). See Figure 2.

The DBR contains the values:

DBR·core which is the absolute core address of the DS.

DBR·L which is the length of the DS.

Segment descriptor word number "s" contains the values:

SDW(s)·core which is the absolute core address of the segment s.

SDW(s)·L which is the length of the segment s.

SDW(s)·acc which describes the access rights for the segment.

SDW(s)·F which is the "missing segment" switch.

A simplified version of the algorithm used by the processor to access the word whose address is $[s, i]$ follows (see Figure 2):

If DBR·L $< s$, generate a trap, or "fault" to the supervisor.

Access SDW(s) at absolute location DBR·core $+ s$.

If SDW(s)·F = ON, generate a *missing segment fault*.

If SDW(s)·L $< i$, generate a fault.

If SDW(s)·acc is incompatible with the requested operation, generate a fault.

Access the word whose absolute address is SDW(s)·core $+ i$.

5.2 Paging

The above description assumes that segments are not paged; in fact, paging is implemented in the 645 hardware. In the Multics implementation, all segments are paged and the page size is always 1,024 words.

Element "i" of a segment is the w^{th} word of the p^{th} page of the segment, "w" and "p" being defined by

$$\begin{cases} w = i \bmod 1{,}024 \\ p = (i - w)/1{,}024 \end{cases}$$

Each segment is referenced by a processor through a *page table* (PT). The PT of a segment is an array of

Fig. 2. Hardware segmentation in the Honeywell 645.

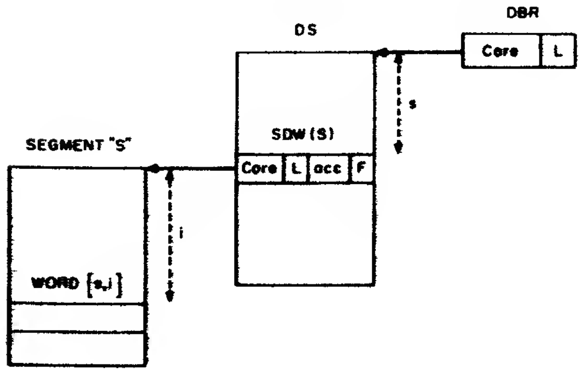
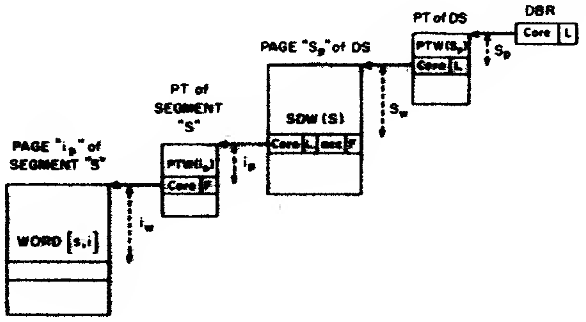


Fig. 3. Hardware segmentation and paging in the Honeywell 645.



physically contiguous words in core memory. Each element of this array is called a *page table word* (PTW). **Page table word number p contains:**

PTW$(p)\cdot$core which is the absolute core address of page number p.

PTW$(p)\cdot$F which is the "missing page" switch.

The meaning of DBR$\cdot$core and SDW$(s)\cdot$core is now:

DBR$\cdot$core = Absolute core address of the PT of the descriptor segment.

SDW$(s)\cdot$core = Absolute core address of the PT of segment number s.

A simplified version of the algorithm used by the processor to access the word whose address is $[s, i]$ is as follows (see Figure 3):

```
If DBR·L $<$ $s$, generate a fault.
Split $s$ into the page number $s_p$ and word number $s_w$.
Access PTW$(s_p)$ at absolute location
    DBR·core + $s_p$.
If PTW$(s_p)$·F = ON, generate a missing page fault.
Access SDW$(s)$ at absolute location
    PTW$(s_p)$·core + $s_w$.
If SDW$(s)$·F = ON, generate a missing segment fault.
If SDW$(s)$·L $<$ $i$, generate a fault.
If SDW$(s)$·acc is incompatible with the requested
    operation, generate a fault.
Split $i$ into the page number $i_p$ and word number $i_w$.
Access PTW$(i_p)$ at absolute location
    SDW$(s)$·core + $i_p$.
If PTW$(i_p)$·F = ON, generate a missing page fault.
Access the word whose absolute location is
    PTW$(i_p)$·core + $i_w$.
```

In order to reduce the number of processor references to core storage while performing this algorithm, each processor has a small, high-speed *associative memory* [12] automatically maintained so as to always contain the PTW's and SDW's most recently used by the processor. The associative memory significantly reduces the number of additional memory requests required during address preparations.

6. Multics Processes and the Multics Supervisor

A process is generally understood as being a program in execution. A process is characterized by its state-word defining, at any given instant, the history resulting from the execution of the program. It is also characterized by its *address space*. The address space of a process is the set of processor addresses that the process can use to reference information in memory. In Multics, any information that a process can reference by an address of the form (segment number, word number) is said to be in the address space of the process. There is a one-to-one correspondence between Multics processes and address spaces. Each process is provided with a private descriptor segment which maps segment numbers into core memory addresses and with a private table which maps symbolic segment names into segment numbers. This table is called the Known Segment Table (KST).

The Multics supervisor could have been written so as not to use segment addressing of course; but organizing the supervisor into procedures and data segments permits one to use, in the supervisor, the same conventions that are used in user programs. For instance, the call-save-return conventions [7] made for user programs can be used by the supervisor; the standard way to manufacture pure procedures in a user program is also used extensively in the supervisor. A less visible advantage of segmentation of the supervisor is that some supervisory facilities provided for the management of user segments can also be applied to supervisor segments; for example, the demand paging facility designed to automatically load pages of user segments can also be used to load pages of supervisor segments. As a result, a large portion of the supervisor need not reside permanently in core.

Unlike most supervisors, the Multics supervisor does not operate in a dedicated process or address space. Instead, the supervisor procedure and data segments are shared among all Multics processes. Whenever a new process is created, its descriptor segment is initialized with descriptors for all supervisor segments allowing the process to perform all of the basic supervisory functions for itself. The execution of the supervisor in the address space of each process facilitates communication between user procedures and supervisor procedures. For example, the user can call a supervisor procedure as if he were calling a normal user procedure. Also, the sharing of the Multics supervisor facilitates simultaneous execution, by several processes, of supervisory functions, just as the sharing of user procedures facilitates the simultaneous execution of functions written by users.

Since supervisor segments are in the address space of each process, they must be protected against unauthorized references by user programs. Multics provides the user with a ring protection mechanism [13] which segregates the segments in his address space into several sets with different access privileges. The Multics supervisor takes advantage of the existence of this mechanism and uses it, rather than some other special mechanism to protect itself.

7. Segment Attributes

7.1 Directory Hierarchy

The name of a segment and its attributes are associated in a catalogue. Conceptually this catalogue consists of a table with one entry for each segment in the system. An *entry* contains the name of the segment and all its attributes: length, memory address, list of users allowed to use the segment with their respective access rights, date and time the segment was created, etc.

In Multics, this catalogue is implemented as several segments, called directories, organized into a tree structure. A *segment name* is a list of subnames reflecting the position of the entry in the tree structure, with respect to the beginning, or root directory (ROOT) of the tree. By convention, subnames are separated by the character ">". Each subname is called an *entryname* and the list of entrynames is called a *pathname*. An entryname is unique in a given directory and a pathname is unique in the entire directory hierarchy. Because of its property of uniquely identifying a segment in the directory hierarchy, the pathname has been chosen as the symbolic name by which the Multics user must reference a segment. There are two types of directory entries, branches and links. A *branch* is a directory entry which contains all attributes of a segment while a *link* is a directory entry which contains the pathname of another directory entry. A more detailed description of the directory hierarchy and of the use of links is given by Daley and Neumann [6].

7.2 Operations on Segment Attributes

Supervisor primitives perform all operations on segment attributes. There is a set of primitives available to the user which allow him, for example, to create a segment, delete a segment, change the entryname of a directory entry, change the access rights of a segment, list the segment attributes contained in a directory, etc.

Creating a segment whose pathname is ROOT > A > B > C (see Figure 4) consists basically of the following steps:

Check that entryname C does not already exist in the directory ROOT > A > B.

Allocate space for a new branch in directory ROOT > A > B.

Store in the branch the following items:

The entry name C.

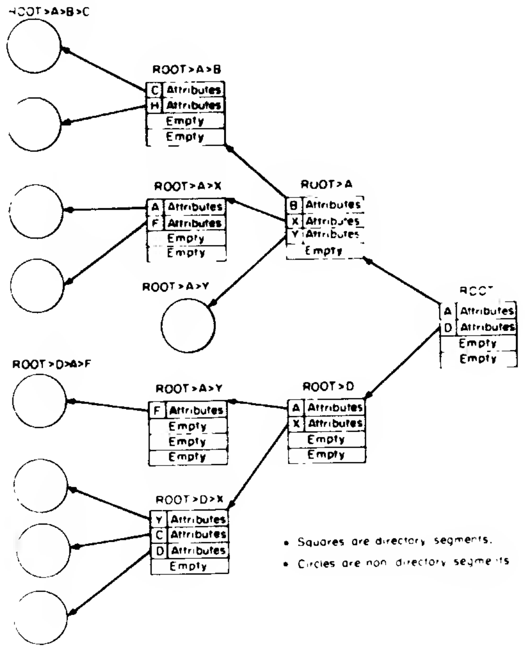
The segment length, initialized to zero.

The access list, given by the creator.

The segment map, consisting of an array of secondary memory addresses, one for each page of the segment. The maximum length of a segment in Multics being 64 pages, the segment map for any segment contains 64 entries. Since the segment length is still zero, each entry of the segment map is initialized with a "null" address, showing that no secondary memory has been assigned to any potential page of the segment.

The segment status "inactive," meaning that there is no page table for this segment. The segment status, which may be either "active" or "inactive" is indicated by the *active switch*.

Fig. 4. Directory hierarchy.



8. Segment Accessing

Although the creation of a segment initializes its attributes, additional supervisor support is required to make the segment accessible to the processor when a user program references the segment by symbolic name.

8.1 Symbolic Addressing Conventions

The pathname is the only symbolic name by which a segment can be uniquely identified in the directory hierarchy. However, for user convenience, the system provides a facility whereby a user can reference a segment from his program using only the last entryname of the segment's pathname and supplying the rest of the pathname according to system conventions. This last entry name is called the *reference name.*

When a process executes an instruction which attempts to access a segment by means of its reference name, the Multics dynamic linking facility [7] is automatically invoked. The dynamic linker determines the missing part of the pathname according to the above-mentioned system conventions. These conventions are called *search rules* and may be regarded as a list of directories to be searched for an entryname matching the specified reference name. When this entryname is found in a directory, the directory pathname is prefixed to the reference name yielding the required pathname. The dynamic linker, using the "Make Known" module (Section 8.2), then obtains a segment number by which the referenced segment will be accessed. Finally it transforms the reference name into this segment number so that all subsequent executions of the instruction in this process access the segment directly by segment number. Further details are given by Daley and Dennis [7].

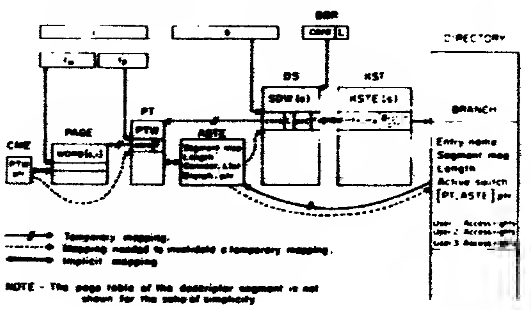
8.2 Making a Segment Known to a Process

Each time a segment is referenced in a process by its pathname, either explicitly or as the result of the evaluation of a reference name by the dynamic linking facility, the pathname must be translated into a segment number in order to permit the processor to address the segment for this process. This translation is done by the supervisor using the KST associated with the process. The KST is an array organized such that entry number "*s*", KSTE(*s*), contains the pathname associated with segment number "*s*". See Figure 5.

If the association (pathname, segment number) is found in the KST of the process, the segment is said to be *known* to the process and the segment number can be used to reference the segment.

If the association (pathname, segment number) is not found in the KST, this is the first reference to the segment in the process and the segment must be made known. A segment is made known by assigning an unused segment number "*s*" in the process and by recording the pathname in KSTE(*s*) to establish the pair (pathname, segment number) in the KST of the process. The directory hierarchy is also searched for this pathname and a pointer to the corresponding branch is entered in KSTE(*s*) for later use (Section 8.3.).

Fig. 5. Basic tables used to implement the Multics virtual memory.



The per-process association of pathname and segment number is used in the Multics system because it is impossible to assign a unique segment number to each segment. The reason is that the number of segments in the system will nearly always be larger than the number of segment numbers available in the processor.

When a segment is made known to a process by segment number "*s*," its attributes are not placed in SDW(*s*) of the descriptor segment of that process. SDW(*s*) having been initialized with the missing segment switch ON, the first reference in this process to that segment by segment number "*s*" will cause the processor to generate a trap. In Multics this trap is called a "missing segment fault" and transfers control to a supervisor module called the segment fault handler.

8.3 The Segment Fault Handler

When a missing segment fault occurs, control is passed to the segment fault handler to store the proper segment attributes in the appropriate SDW and set the missing segment switch OFF in the SDW.

These attributes, as shown in Figure 3, consist of the page table address, the length of the segment, and the access rights of the user with respect to the segment. The information initially available to the supervisor upon occurrence of a missing segment fault is the segment number "s."

The only place where the needed attributes can be found is in the branch of the segment. Using the segment number "*s*", the supervisor can locate the KST entry associated with the faulting segment; it can then find the required branch since a pointer to the branch has been stored in the KST entry when the segment was made known to this process (Section 8.2).

Using the active switch (Figure 5) in the branch, the supervisor determines whether there is a page table for this segment. Recall that this switch was initialized in the branch at segment creation time. If there is no page table, one must be constructed. A portion of core memory is permanently reserved for page tables. All page tables are of the same length and the number of page tables is determined at system initialization.

The supervisor divides page tables into two lists: the used list and the free list. Manufacturing a page table (PT) for a segment could consist only of selecting a PT from the free list, putting its absolute address in the branch and moving it from the free to the used list. If this were actually done, however, the servicing of each missing page fault would require access to a branch since the segment map containing secondary storage addresses is kept there (Figure 5). Since it is impractical for all directories to permanently reside in core, page fault handling could thereby require a secondary storage access in addition to the read request required to transport the page itself into core. Although this mechanism works, efficiency considerations have led to the "activation" convention between the segment fault handler and the page fault handler.

Activation. A portion of core memory is permanently reserved for recording attributes needed by the page fault handler, i.e. the segment map and the segment length. This portion of core is referred to as the *active segment table* (AST). There is only one AST in the system and it is shared by all processes. The AST contains one entry (ASTE) for each PT. A PT is always associated with an ASTE, the address of one implying the address of the other. They may be regarded as a single entity and will be referred to as the (PT, ASTE) of a segment. The used list and free list mentioned above are referred to as the (PT, ASTE) *free list* and the (PT, ASTE) *used list.*

A segment which has a (PT, ASTE) is said to be *active*. Being active or not active is an attribute of the segment and is recorded in the branch using the active switch.

When the active switch is ON, both the segment map and the segment length are no longer in the branch but are to be found in the segment's (PT, ASTE) whose address was recorded in the branch during "activation" of the segment.

To activate a segment, the supervisor must:

Find a free (PT, ASTE). (Assume temporarily that at least one is available).

Move the segment map and the segment length from the branch into the ASTE.

Set the active switch ON in the branch.

Record the pointer to (PT, ASTE) in the branch.

By pairing an ASTE with a PT in core, the segment fault handler has guaranteed that all segment attributes needed by the page fault handler are core-resident, permitting more efficient page fault servicing.

Connection. Once the segment is active, the corresponding SDW must be "connected" to the segment. To connect the SDW to the segment the supervisor must:

Get the absolute address of the PT, using the (PT, ASTE) pointer kept in the branch, and store it in SDW.

Get the segment length from the ASTE and store it in the SDW.

Get the access rights for the user from the branch and store them in the SDW.

Turn off the missing segment switch in the SDW.

Having defined activation and connection, segment fault handling can now be summarized as:

Use the segment number s to access the KST entry.

Use the KST entry to locate the branch.

If the active switch in the branch is OFF, activate the segment.

Connect the SDW.

Note that the active switch and the (PT, ASTE) pointer in the segment branch "automatically" guarantee segment sharing in core since all SDW's describing a given segment will point to the same PT.

Once the segment and its SDW have been connected, the hardware can access the appropriate page table word. If the page is not in core, a missing page fault occurs, transferring control to the supervisor module called the page fault handler.

8.4 The Page Fault Handler

When a page fault occurs the page fault handler is given control with the PT address and the page number of the faulting page. The information needed to bring the page into core memory is the address of a free block of core memory into which the page can be moved and the address of the page in secondary memory. The term *page frame* is also used to denote a block of core memory which holds a page of information [9].

A free block of core must be found. This is done by using a data base called the *core map*. The core map is an array of elements called *core map entries* (CME). The n^{th} entry contains information about the n^{th} block of core (the size of all blocks is 1,024 words). The supervisor divides this core map into two lists; the *core map used list* and the *core map free list.*

The job of the page fault handler consists of the following steps:

Find a free block of core and remove its core map entry from the free list. (Assume temporarily that the free list is not empty.)

Access the ASTE associated with the PT and find the address in secondary memory of the missing page.

If this address is a "null" address, initialize the block of core with zeros and update the segment length in the ASTE; this action is only taken the first time the page is referenced since the segment was created and provides for the automatic growing of segments. Otherwise issue an I/O request to move the page from secondary memory into the free block of core and wait for completion of the request via a call to the "traffic controller" [14] which is responsible for processor multiplexing.

Store the core address in the PTW, remove the fault from the PTW, and place the core map entry in the used list.

8.5 Page Multiplexing

There are many more pages in virtual memory than there are blocks of core in the real memory; therefore, these blocks must be multiplexed among all pages. In the description of page fault handling it was assumed that a free block of core was always available. In order to insure that this is nearly always true, the page fault handler, upon removing a free block from the core map free list, examines the number of remaining free list entries; if this number is less than a preset minimum value, a page removal mechanism is invoked a sufficient number of times to ensure a nonempty core map free list in all but the most unusual cases. A nonempty core map free list eliminates waiting for page removal during the handling of a missing page fault.

To get a free block of core, the page removal mechanism may have to move a page from core to secondary memory. This requires: (a) an algorithm to select a page to be removed; (b) the address of the PTW which holds the address of the selected page, in order to set a fault in it; and (c) a place to put the page in secondary memory.

The selection algorithm is based upon page usage. It is a particularly easy-to-implement version [4] of the "least-recently-used" algorithm [1, 8]. The hardware provides valuable assistance by, each time a page is referenced, setting ON a bit, called the *used bit*, in the corresponding PTW. The selection algorithm will not be described in detail here. However, it should be noted that candidates for removal are those pages described in the core map used list; therefore, each core map entry which appears in the used list must contain a pointer to the associated PTW (Figure 5) in order to permit examination of the used bit. The action of storing the PTW pointer in the core map entry must be added to the list of actions taken by the page fault handler when a page is moved into core (Section 8.4.).

Once the supervisor has selected the page to be removed, it takes the following steps:

Set the missing page switch ON in the PTW.

If no secondary memory has been assigned yet for this page, i.e. the segment map entry for this page holds a "null" address, assign a block of secondary memory and store its address in the segment map entry.

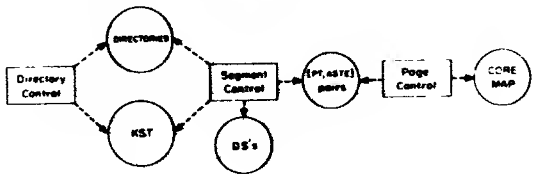
Issue an I/O request to move the page to secondary storage.

Upon completion of the I/O request, move the core map entry describing the freed block of core from the core map used list to the core map free list. This may be done in another process upon noticing the completion of the I/O request.

8.6 (PT, ASTE) Multiplexing

Core blocks can be multiplexed only among pages of active segments. The number of concurrently active segments is limited to the number of (PT, ASTE) pairs, which is, by far, smaller than the total number of segments in the virtual memory. Therefore (PT, ASTE) pairs must be multiplexed among all segments in the virtual memory.

Fig. 6. Supervisor functional modules and data bases.



When segment activation was described, a (PT, ASTE) pair was assumed available for assignment. In fact, this is not always the case. Making one segment active may imply making another segment inactive, thereby disassociating this other segment from its (PT, ASTE). Since all processes sharing the same segment will have the address of the PT in an SDW, it is essential to invalidate this address in all SDW's containing it before removing the page table.

This operation requires: (a) an algorithm to select a segment to be deactivated; (b) knowing all SDW's that contain the address of the page table of the selected segment, in order to invalidate this address; (c) moving the attributes contained in the ASTE back to the branch; and (d) changing the status of the segment from active to inactive in the branch.

The selection algorithm for deactivation, like the selection algorithm for page removal, is based on usage. When the last page of a segment is removed from core, the segment becomes a candidate for deactivation. The algorithm selects for deactivation the segment which has had no pages in core for the longest period of time, i.e. the segment which has been least recently used. Since the number of (PT, ASTE) pairs substantially exceeds the number of pageable blocks of core, it is always possible to find an active segment with no pages in core.

The ASTE must provide all the information needed for deactivating a segment. This means that during activation and connection, this information must be made available. During activation, a pointer to the branch must be placed in the ASTE; during connection, a pointer to the SDW must be placed in the ASTE. Since more than one SDW is connected to the same PT when the segment is shared by several processes, the supervisor must maintain a list of pointers to all connected SDW's. This list is called a connection list. See Figure 5.

After the selection algorithm chooses a (PT, ASTE) to be freed, the disassociation of the segment from its

(PT, ASTE) is done in two steps: *disconnection* and *deactivation*.

Disconnection consists of storing a segment fault in each SDW whose address appears in the connection list in the ASTE. Deactivation consists of moving the segment map and the segment length from the ASTE back to the branch, resetting the active switch in the branch, and putting the (PT, ASTE) in the free list.

9. Structure of the Supervisor

Up to now supervisor functions have been described, but not the supervisor structure. In this section, the different components of the supervisor are presented and the ability of portions of the supervisor to utilize the virtual memory is discussed.

9.1 Functional Modules

Three functional modules can be identified in the supervisor described in Section 8; they are called *directory control* (DC), *segment control* (SC), and *page control* (PC).

DC performs all operations on segment attributes; it also maps pathnames into segment numbers in the KST of the executing process. Data bases used by a process executing DC procedures are the directories and the KST of the process (Figure 6).

SC performs segment fault handling. Data bases used by a process executing SC procedures are directories, the KST of the process, descriptor segments and (PT, ASTE) pairs.

PC performs page fault handling. Data bases used by a process executing PC procedures are (PT, ASTE) pairs and the core map.

9.2 Use of PC in the Supervisor

One can observe that the page fault handler need not know if a missing page belongs to a user segment or to a supervisor segment; it only expects to find the information it requires in the (PT, ASTE) of the segment to which the missing page belongs. Therefore, if all segments used in SC and DC are always active, then their pages need not be in core since PC can load them when they are referenced.

In order to make use of PC in the rest of the supervisor the following (temporary) assumption must be made.

Assumption 1
(a) All segments used in PC are always in core and are connected to the descriptor segment of each process.
(b) All segments used in SC and DC are always active and are connected to the descriptor segment of each process.

9.3 Use of SC in the Supervisor

Assumption 1 is satisfactory in the Multics implementation *except for directories.*

The number of directory segments in the system may be very large and keeping them always active is not a realistic approach, since a large number of (PT, ASTE) pairs would have to be permanently assigned to them. It would be desirable to use SC to activate and connect directory segments only as needed.

A necessary condition for handling a segment fault for segment x in a process is that segment x be known to that process. Assuming that all directories are known to all processes, but not necessarily active, reference to a directory x may cause a segment fault. When handling this fault, the segment fault handler must reference the parent directory of segment x, where the branch for x is located. This reference to the parent of x could, in turn, cause a recursive invocation of the segment fault handler. These recursive invocations can propagate from directory to parent directory up to the root. If the root directory is always active and connected to each process, then the recursion is guaranteed to be finite and a segment fault for any directory can be handled.

The first assumption can be replaced by the following more satisfactory assumption (again temporary).

Assumption 2
(a) All segments used in PC are always in core and are connected to the descriptor segment of each process.
(b) All nondirectory segments used in SC and DC are always active and are connected to the descriptor segment of each process.
(c) The root directory is always active and connected to each process.
(d) All directories are always known to each process.

9.4 Use of the Make Known Facility in the Supervisor

However, it is unsatisfactory to keep all directories known to all processes because of the space that would be required in each KST. It would be more attractive if a directory could be made known to a process only when needed by the process.

Making a segment x known implies searching for its pathname in the KST. If not found, the parent of x must first be made known and so on up to the root. If the root directory is always known to all processes, then any directory can be made known to a process by calling recursively the Make Known facility of the supervisor.

Assumption 2 will now be replaced by the final assumption:

Final Assumption
(a) All segments used in PC are always in core and are connected to the descriptor segment of each process.
(b) All nondirectory segments used in SC and DC are always active and are connected to the descriptor segment of each process.
(c) The root directory is always active and connected to each process.
(d) The root directory is always known to each process.

Given the above assumption, supervisor segments, as well as user segments, can be stored in the virtual memory that the supervisor provides.

10. Summary

The most important points discussed in this paper are summarized below. They are grouped into two classes: the point of view of the user of the virtual memory, and the point of view of the supervisor itself.

User Point of View

The Multics virtual memory can contain a very large number of segments that are referenced by symbolic names.

Segment attributes are stored in special segments called directories, which are organized into a tree structure; by a naming convention known to the user, the symbolic name of a segment must be the pathname of the segment in the directory tree structure.

Any operation on directory segments must be done by calling the supervisor.

Any operation on a nondirectory segment can be done directly in accordance with the access rights that the user has for the segment; any word of any segment which resides in the virtual memory can be referenced with a pair (pathname, *i*) by the user.

Supervisor Point of View

The supervisor must simulate a large segmented memory which is directly addressable by symbolic name and such that any access to the memory is submitted to access rights checking.

The supervisor maintains a directory tree where it stores all segment attributes. It can retrieve the attributes of a segment, given the pathname of that segment.

The supervisor itself is organized into segments and runs in the address space of each user process.

Any segment, be it a directory or a nondirectory segment, is identified by its pathname but can be accessed only using a segment number. For each segment name the supervisor must assign a segment number by which the processor will address the segment in the process.

The processor accesses a word of a segment through the appropriate SDW and PTW, subject to the access rights recorded in the SDW.

A segment fault is generated by the processor whenever the page table address or access rights are missing in the SDW. The supervisor then, using the KST entry as a stepping stone, accesses the branch where it finds the needed information. If a PT is to be assigned, the supervisor may have to deactivate another segment.

A page fault is generated by the processor whenever a PTW does not contain a core address. The supervisor then, using the ASTE associated with the PT, moves the missing page from secondary storage to core. This may require the removal of another page.

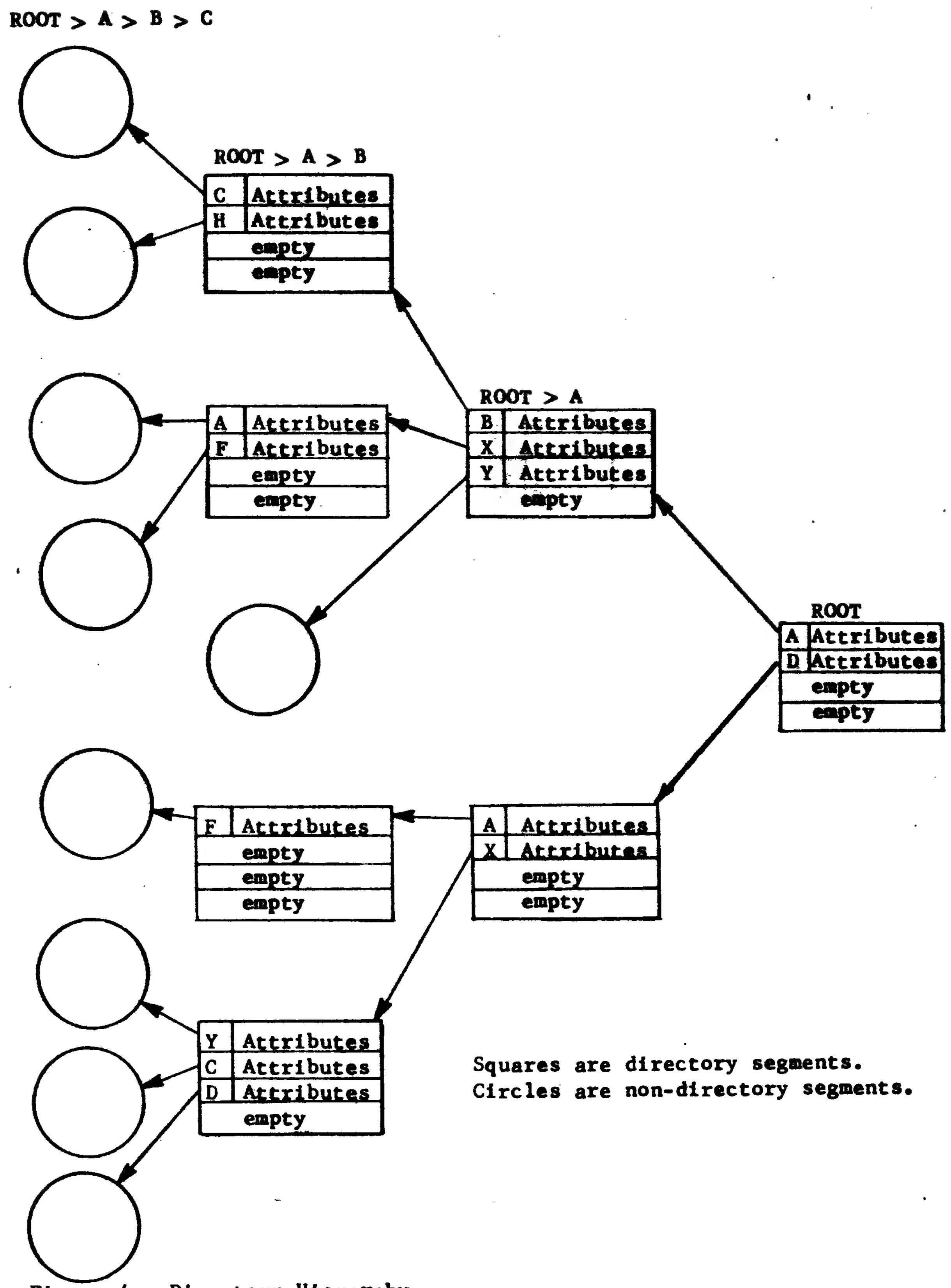
Acknowledgments. This paper would be incomplete without acknowledgment of the people who worked so hard to build the virtual memory supervisor portion of Multics. Special mention goes to G.F. Clancy, M.R. Thompson, and S.H. Webber who, under the design leadership of R.C. Daley, have been involved in a major portion of the design and implementation effort. They were aided in earlier designs and implementations by C.A. Cushing, S.M. Jones, G.B. Krekeler, N.I. Morris, P.G. Neumann, R.K. Rathbun, J.D. Van Hausan, M.R. Wagner, and L.D. Whitehead. Recent implementations have also benefited from the contributions of S.D. Dunten and M.C. Turnquist. Contributions in the form of analyses and discussions have been made by F.J. Corbató, E.L. Glaser, J.H. Saltzer, and V.A. Vyssotsky.

Finally, our thanks go to P.A. Belmont, M.A. Meer, and D.L. Stone, who participated in studies leading to this formalized description of the Multics virtual memory.

Received April 1970; revised July 1971

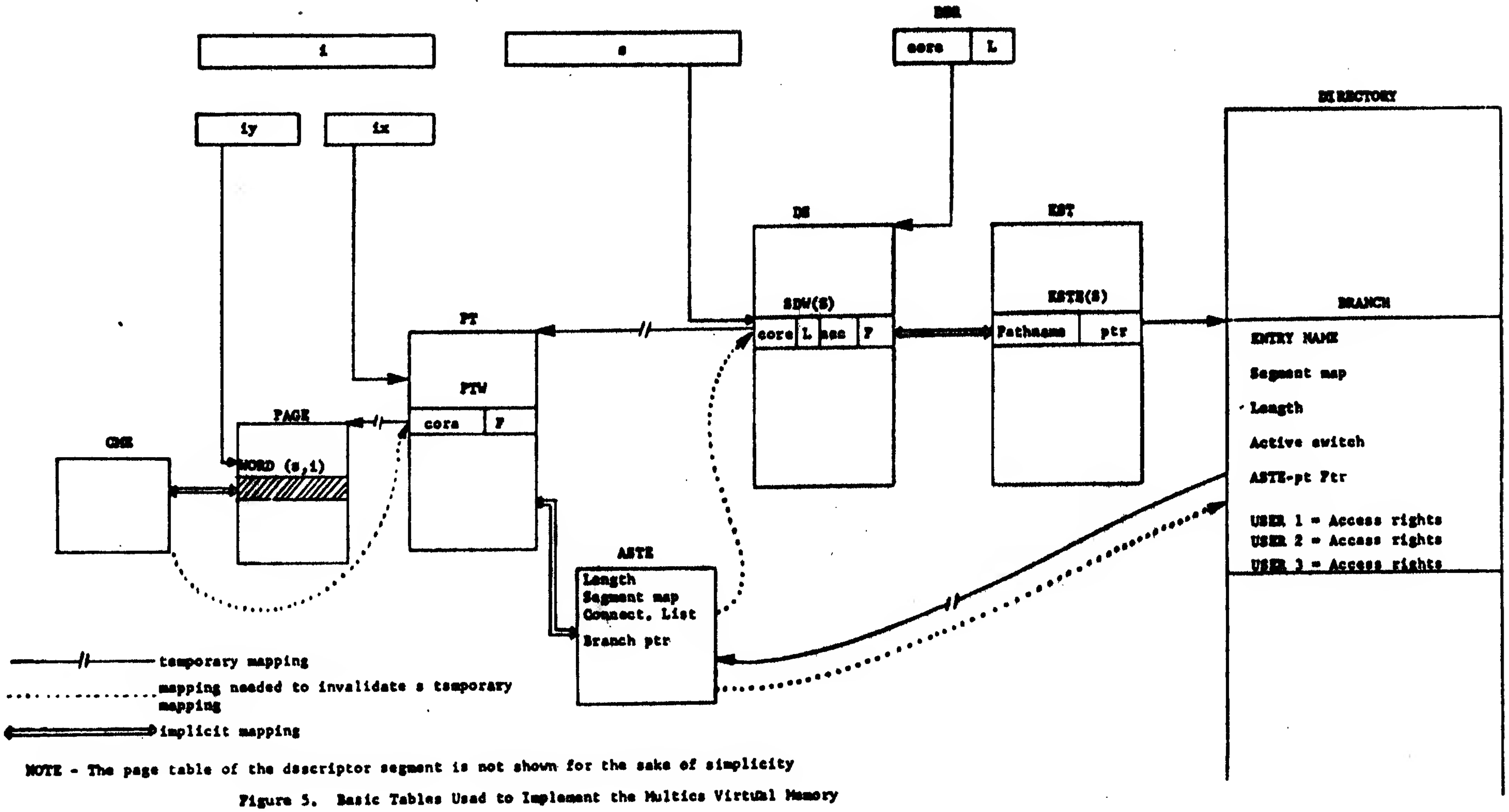
References

1. Belady, L.A. A study of replacement algorithms for a virtual-storage computer. *IBM Systems J.5*, 2 (1966), 78–101.
2. Comfort, W.T. A computing system design for user service. Proc. AFIPS 1965 FJCC, Vol. 27, Pt. 1, Spartan Books, New York, pp. 619–628.
3. Corbató, F.J., and Vyssotsky, V.A. Introduction and overview of the Multics system. Proc. AFIPS 1965 FJCC, Vol. 27, Pt. 1. Spartan Books, New York, pp. 185–196.
4. Corbató, F.J. A paging experiment with the Multics system. Included in a Festschrift published in honor of Prof. P.M. Morse. MIT Press, Cambridge, Mass., 1969.
5. Crisman, P.A. Ed. *The Compatible Time-Sharing System*: A Programmer's Guide, 2nd Ed., MIT Press, Cambridge, Mass., 1965.
6. Daley, R.C., and Neumann, P.G. A general-purpose file system for secondary storage. Proc. AFIPS 1965 FJCC, Vol. 27, Pt. 1. Spartan Books, New York, pp. 213–229.
7. Daley, R.C., and Dennis, J.B. Virtual memory, processes, and sharing in Multics. *Comm. ACM 11*, 5 (May 1968), 306–312.
8. Denning, P.J. The working set model for program behavior. *Comm. ACM 11*, 5 (May 1968), 323–333.
9. Denning, P. J. Virtual memory. *Computing Surveys 2*, 3 (Sept. 1970), 153–189.
10. Dennis, J.B. Segmentation and the design of multiprogrammed computer systems. *J.ACM 12*, 4 (Oct. 1965), 589–602.
11. Fotheringham, J. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Comm. ACM 4*, 10 (Oct. 1961), 435–436.
12. Glaser, E.L., Couleur, J.F., and Oliver, G.A. System design of a computer for time sharing applications. Proc. AFIPS 1965, FJCC, Vol. 27, Pt. 1. Spartan Books, New York, pp. 197–202.
13. Graham, R.M. Protection in an information processing utility. *Comm. ACM 11*, 5 (May 1968), 365–369.
14. Saltzer, J. H. Traffic Control in a Multiplexed Computer System. Tech. Rep. No. MAC-TR-30 (Ph.D. Thesis), Project MAC, MIT, Cambridge, Mass., 1964.
15. The Descriptor—A definition of the B5000 Information Processing System. Burroughs Corp., Detroit, Mich., 1961.



Squares are directory segments.
Circles are non-directory segments.

Figure 4. Directory Hierarchy



NOTE - The page table of the descriptor segment is not shown for the sake of simplicity

Figure 5. Basic Tables Used to Implement the Multics Virtual Memory

Virtual Memory, Processes, and Sharing in Multics

by R.C. Daley and J.B. Dennis. Reprinted from Communications of the ACM 11, 5, May, 1968, pp. 306-312, with permission. Copyright 1968 by the Association for Computing Machinery.

This early paper introduced the concept of a virtual memory which contains all on-line storage, and explains the hardware addressing structure which is used to support it. The remainder of the paper then explores the properties and mechanisms necessary to permit dynamic linking of procedures and data. The paper does not emphasize the value to the user of this feature. Briefly, dynamic linking eliminates the need to collect together all the parts of a program before execution; it is especially helpful during debugging of a new program. A more extensive discussion of the usefulness of this feature is found in MPM Introduction Chapter Four.

It may help, when reading the discussion of dynamic linking, to realize that stored as part of every pure procedure is a prototype linkage section for that procedure. When the procedure is first linked to, the dynamic linker copies this prototype linkage section into the linkage area for the process, and this copy is the linkage section referred to in the paper. Note that the word "linking" is a local piece of jargon, which has a meaning approximately the same as "binding" in most recent literature on languages and linguistics.

The call-save-return mechanism described in the paper was the first one used in Multics, and is quite different from the one implemented with special hardware in the current Honeywell 6180 system. However, the mechanism described is functionally equivalent to the current one, and it is quite instructive to compare the description here with that provided in the Subsystem Writers' Guide, to gain insight into the intrinsic operations being performed. Probably the most important difference between the two mechanisms is that the older one described in this paper required that the linkage section contain instructions to be executed as part of the subroutine entry sequence. In the newer technique the linkage section contains only indirect addresses. As a result, the segment containing the linkage section no longer requires "execute" permission, and wild transfers to that segment are thus trapped immediately as errors.

Virtual Memory, Processes, and Sharing in MULTICS

Robert C. Daley and Jack B. Dennis
Massachusetts Institute of Technology, Cambridge, Massachusetts

Some basic concepts involved in the design of the MULTICS operating system are introduced. MULTICS concepts of processes, address space, and virtual memory are defined and the use of paging and segmentation is explained. The means by which users may share procedures and data is discussed and the mechanism by which symbolic references are dynamically transformed into virtual machine addresses is described in detail.

KEY WORDS AND PHRASES: virtual memory, information sharing, shared procedures, data sharing, dynamic linking, segmentation, paging, multiprogramming, storage management, storage hierarchies, file maintenance
CR CATEGORIES: 3.73, 4.32

Presented at an ACM Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1-4, 1967; revised December, 1967.
This paper is based on notes prepared by J. Dennis for the University of Michigan Summer Conference on Computer and Program Organization, June 1966.
The work reported herein was supported in part by Project MAC, an M.I.T. research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction of this report, in whole or in part, is permitted for any purpose of the United States Government.

Introduction

In MULTICS [1] (*Mult*iplexed *I*nformation and *C*omputing *S*ervice), fundamental design decisions were made so the system would effectively serve the computing needs of a large community of users with diverse interests, operating principally from remote terminals. Among the objectives were these three:

(1) To provide the user with a large machine-independent virtual memory, thus placing the responsibility for the management of physical storage with the system software. By this means the user is provided with an address space large enough to eliminate the need for complicated buffering and overlay techniques. Users, therefore, are relieved of the burden of preplanning the transfer of information between storage levels, and user programs become independent of the nature of the various storage devices in the system.

(2) To permit a degree of programming generality not previously practical. This includes the ability of one procedure to use another procedure knowing only its name, and without knowledge of its requirements for storage, or the additional procedures upon which it may in turn call. For example, a user should be able to initiate a computa-

tion merely by specifying the symbolic name of a procedure at which the computation is to start and by allowing additional procedures and data to be provided automatically when and if they are needed.

(3) To permit sharing of procedures and data among users subject only to proper authorization. Sharing of procedures in core memory is extremely valuable in a multiplexed system so that the cluttering of auxiliary storage with myriad copies of routines is avoided, and so unnecessary information transfers are eliminated. The sharing of data objects in core memory is necessary to permit efficient and close interaction between processes.

These objectives led to the design of a computer system [6] (the General Electric Model 645) embodying the concepts of paging [8] and segmentation [3] on which the initial implementation of MULTICS will run.

In this paper we explain some of the more fundamental aspects of the MULTICS design. The concepts of "process" and "address space" are defined, some details of the addressing mechanism are given, and the mechanism by which "dynamic linking" is accomplished is explained.

Concepts of Process and Address Space

Several interpretations of the term "process" have come into recent use. The most common usage applies the term to the activity of a processor in carrying out the computation specified by a program [4, 5]. In MULTICS, the concept of process is intimately connected with the concept of address space. Processes stand in one-to-one correspondence with virtual memories. Each process runs in its own address space, which is established independently of other address spaces. Processes are run on a processor at the discretion of the *traffic controller* module of the supervisor.

The virtual memory (or address space) of a MULTICS process is an ordered set of as many as 2^{14} *segments* each consisting of as many as 2^{18} 36-bit *words*. The arguments for providing a generous address space having this structure have been given by Dennis [3]. Briefly, the motivation is to avoid the necessity of procedure overlays or the movement of data within the address space, which generally lead to naming conflicts and severe difficulties in sharing information among many processes.

Each segment is a logically distinct unit of information having attributes of length and access privilege and may grow or shrink independently of other segments in the system. For present purposes, we consider two segment types: (1) data, and (2) procedure. A segment is treated as procedure if it is intended to be accessed for instruction fetch by a processor. Other segments (including, e.g., a source program file) are considered to be data. Instruction fetch references to procedure segments are allowed, as are internal data reads. Writing into a procedure segment is normally considered invalid and is prohibited by the system. (In certain cases, reading of a procedure segment by another procedure may also be prohibited while execution is allowed.) Thus procedure segments are nonself-modifying or *pure* procedures. Instruction fetches from data segments are invalid, and any data segment may be write protected. The overall design of MULTICS protection mechanisms is discussed by Graham [7].

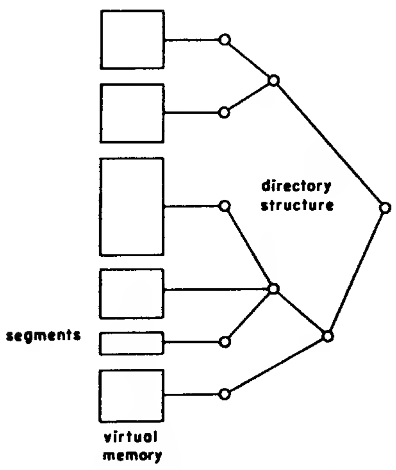


FIG. 1. Virtual memory in a MULTICS process

The size of address space provided to processes makes it feasible to dispense with files as a separate mechanism for addressing information held in the computer system. No distinction need be drawn between files and segments!

The directory structure [2] is a hierarchical arrangement of directories that associates at least one symbolic name (but perhaps many) with each segment. These names have meaning that is invariant over all processes in existence. Figure 1 portrays the concept of a process as a virtual memory made up of segments selected from the directory structure.

Addressing

The Generalized Address. Each word in the address space of a process is identified by a *generalized address*. As shown in Figure 2, a generalized address consists of two parts—a *segment number* and a *word number*. The addressing mechanisms of the processor are designed so that a process may make effective reference to a word by means of its generalized address when the word has an assigned location in main memory. Together with supervisor software, these mechanisms make reference by generalized address, effective regardless of where the word might reside in the storage hierarchy by placing it in main memory when needed. Thus the generalized address is a *location-independent* means of identifying information. In

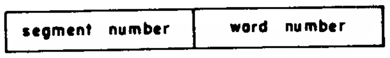


FIG. 2. The generalized address

the following paragraphs we explain how generalized addresses are formed in the processor and give a brief discussion of how they are made effective.

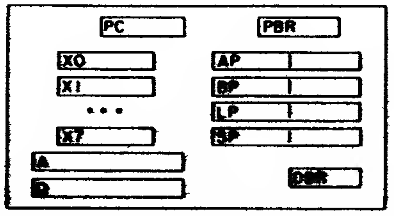


FIG. 3. Processor registers for address formation

Address Formation. Each processor of the computer system (Figure 3) has an accumulator A, a multiplier/quotient Q, eight index registers X0, X1, · · · , X7, and a program counter PC, which serve conventional functions. For the implementation of generalized addressing and intersegment linking, a *descriptor base register*, a *procedure base register*, and four *base pair registers* are included in each processor. The function of the descriptor base register will be discussed in a later paragraph since it does not participate in generalized address formation. The procedure base register always contains the segment number of the procedure being executed. Each of the four base pair registers (called simply base registers in the sequel) holds a complete generalized address (segment number/word number pair) and is named according to its specific function in MULTICS:

base pair	*designation*	*function*
0	<u>ap</u>	argument pointer
1	<u>bp</u>	base pointer
2	<u>lp</u>	linkage pointer
3	<u>sp</u>	stack pointer

The functions of these pointers will become clear when the linkage mechanism is explained.

The instruction format of the processor is given in Figure 4. Instructions are executed sequentially except where a transfer of control occurs. Hence, the program counter is normally advanced by one during the execution of each instruction.

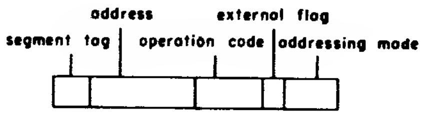


FIG. 4. Instruction format

When the processor requires an instruction word from memory, the corresponding generalized address is the segment number in the procedure base register coupled with the word number in the program counter (Figure 5). For data references, a field in the instruction format called the *segment tag* selects one of the base registers if the *external flag* is on. The effective address computed from the address field of the instruction by the usual indexing procedure is added to the word number portion of the selected base to obtain the desired generalized address. This operation is illustrated by Figure 6 and is used to reference all information outside the current procedure segment. If the *external flag* is off, then the generalized address is the segment number taken from the procedure base register coupled with an effective word number computed as before. This mechanism is used for internal reference by a procedure to fetch constants or for transfer of control.

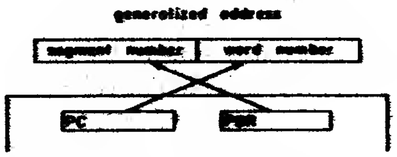


FIG. 5. Address formation for instruction fetch

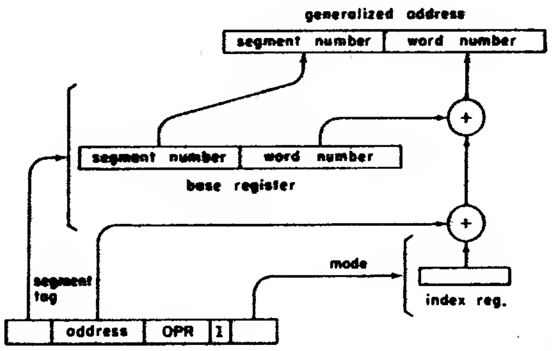


FIG. 6. Address formation for data access

Indirect Addressing. As will be seen when the linkage mechanism is discussed, a method of indirect addressing in terms of generalized addresses is very valuable. In the processor the addressing mode field of instructions may indicate that *indirect addressing* is to be used. In this case, the generalized address, formed as explained above for data references, is used to fetch a pair of 36-bit words which is interpreted as shown in Figure 7. If the address mode field of the first word contains the code <u>its</u> (indirect

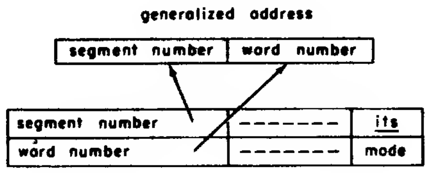


FIG. 7. Interpretation of word pair as indirect address

to segment), the segment number and word number fields are combined to produce a new generalized address. This address is augmented by indexing according to the mode field of the second word of the pair. Further indirect addressing may also be specified.

The Descriptor Segment. Implementation of a memory access specified by a generalized address calls for an associative mechanism that will yield the main memory location of any word within main memory when a segment number/word number combination is supplied. A direct use of associative hardware was impossible to justify in view of the other possibilities available.

The means chosen to implement the generalized address for a process is essentially a two-step hardware table look-up procedure as illustrated by Figure 8. The segment number portion of the generalized address is used as an index to perform a table look-up in an array called the *descriptor segment* of the associated process. This descriptor segment contains a descriptor for each segment that the process may reference by generalized address. Each descriptor contains information that enables the addressing mechanism to locate the segment and information that establishes the appropriate mode ot protection of the segment for this process.

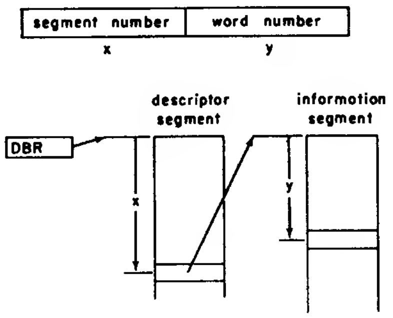


Fig. 8. Addressing by generalized address

The descriptor base register is used by the processor to locate the descriptor segment of the process in execution. Note that since segment numbers and word numbers are nonlocation dependent data, the only location dependent information contained in the processor registers shown in Figure 3 is in the descriptor base register. This fact greatly simplifies the bookkeeping required by the system in carrying out reallocation activity. In fact, switching a processor from one process to another involves little more than swapping processor register status and substituting a new descriptor base.

In practice this implementation requires that segment numbers be assigned starting from zero and continuing successively for the segments of procedure and data required by each process. An immediate consequence is that the same segment will, in general, be identified by *different* segment numbers in different processes.

Paging. Both information segments and descriptor segments may become sufficiently large enough to make paging desirable in order to simplify storage allocation problems in main memory. Paging allows noncontiguous blocks of main memory to be referenced as a logically contiguous set of generalized addresses. The mapping of generalized addresses into absolute memory locations is done by the system and is transparent to the user.

Paging is implemented by means of page tables in main memory which provide for trapping in case a page is not present in main memory. The page tables also contain control bits that record access and modification of pages for use by storage allocation procedures. A small associative memory is built into each processor so that most references to page tables or descriptor segments may be bypassed.

Intersegment Linking and Addressing

The ability of many users to share access to procedure and data information and the power of being able to construct complex procedures by building on the work of others are two prime desiderata of multiprocess computer systems. The potential value of these features to the advancement of computer applications should not be underestimated. The design of a system around the notion of a generalized, location-independent address is an essential ingredient in meeting these objectives. It remains to show how the sharing of data and procedure segments and the building of programs out of component procedure segments can be implemented within the framework of the MULTICS addressing mechanisms just described. In particular we must show how references to external data (and procedure) segments occurring within a shared procedure segment can be correctly interpreted for each of possibly many processes running concurrently.

Requirements. Necessary properties of a satisfactory intersegment addressing arrangement include the following:

(1) Procedure segments must be *pure*; that is, their execution must not cause a single word of their content to be modified.

Pure procedure is a recognized requirement for general sharing of procedure information.

(2) It must be possible for a process to call a routine by its symbolic name without having made prior arrangements for its use.

This means that the subroutine (which could invoke in turn an arbitrarily large collection of other procedures) must be able to provide space for its data, must be able to reference any needed data object, and must be able to call on further routines that may be unknown to its caller.

(3) Segments of procedure must be invariant to the recompilation of other segments.

This requirement has the following implication: The values of identifiers that denote addresses within a segment which may change with recompilation must not appear in the content of any other segment.

Making a Segment Known. Meeting condition (1) requires that a segment be callable by a process even if no position in the descriptor segment of the process has been reserved for the segment. Hence a mechanism is provided in the system for assigning a position in the descriptor segment (a segment number) when the process first makes reference to the segment by means of its symbolic name. We call this operation making the segment *known* to the process. Once a segment is known, the process may reference it by its segment number.

The pattern of descriptor segment assignment will be different for each process. Therefore it is not possible, in general, for the system to assign a unique segment number to a shared routine or data object. This fact is a major consideration in the design of the linking mechanism. In the following paragraphs we describe a scheme for implementing the linkage of segments that meets the requirements stated above.

It is worth emphasizing that this discussion has nothing to do with the memory management problem that the supervisor faces in deciding where in the storage hierarchy information should reside. All information involved in the linkage mechanism is, as will be seen, referenced by generalized addresses which are made effective by the mechanisms described earlier. The fact that pages of the segments referred to in the following discussion may be in or out of main memory at the time a process requires access to them is irrelevant.

Linkage Data. Before a segment becomes known to a process the segment may only be referenced by means of a symbolic *path name* [2] which permanently identifies the segment within the directory structure. Since the segment number used to reference a particular segment is process dependent, segment numbers may not appear internally in pure procedure code. For this reason, a segment is identified within a procedure segment by a symbolic *segment reference name.* Before a procedure can complete an external segment reference, the reference name must be translated into a path name by means of a directory searching algorithm and the desired segment made known to the process. Once the segment has become known to the process, we wish to substitute the efficient addressing mechanism based on the generalized address for the time-consuming operation of searching the directory structure.

Consider a procedure segment P that makes reference to a word at location x within data segment D, as illustrated in Figure 9. In assembly language this would be written as:

$$\text{OPR} \quad \langle D\rangle \,|\, [x]$$

The angle brackets indicate that the enclosed character string is the reference name of some segment. This name will be used to search the directory structure the first time segment P is referenced by a process. The square brackets indicate that the enclosed character string is a symbolic address within an external segment. Since by requirement (3) we wish segment P to be invariant to recompilation of D, only the symbolic address [x] may appear in P. Furthermore, we wish to delay the evaluation of [x] until a reference to it is actually made in the running of a process.

The following problem arises: Initially process α in executing procedure P may reference $\langle D\rangle \,|\, [x]$ only by symbolic segment name and symbolic external address. After segment D has been made known to process α, and a first reference has been effected, we wish to make further references by the generalized address $d \,\#_{\alpha}|x$. The question is: How can we make the transition from symbolic reference to generalized addressing without altering the content of segment P?

It should be clear that a change must be made *some* place that can effect the change in addressing mechanism. Further, the data that is changed must participate in *every* reference to the information. We call the information that is altered in value to make this transition the *link data* for linking segment P to symbolic address

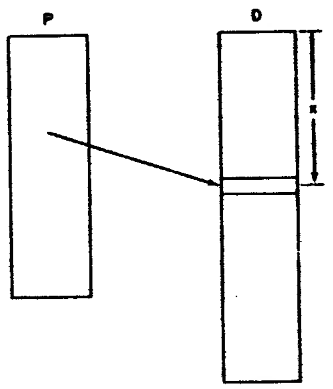


Fig. 9. An intersegment reference by procedure P

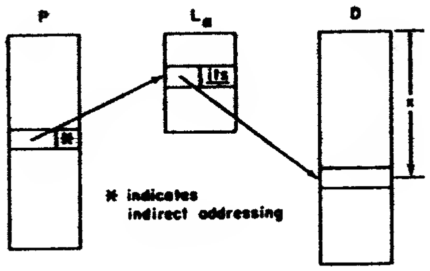


Fig. 10. Linkage of P to D | x for process α

$\langle D\rangle|[x]$ in process α. The collection of link data for all external references originating in segment P is called the *linkage section* of procedure P.

Link data is private data of its process because whether P is linked to D|x for process α is entirely independent of whether the same is true for any other process. Therefore, whenever a procedure segment is made known to a process, a copy of the procedure's linkage section is made as a segment within that process. In certain cases the linkage sections of several procedures are combined into a single linkage segment private to the process.

Linking. Figure 10 shows segments P, D and the linkage section L_α for P in process α. To implement reference to D|x from within segment P will require two references by generalized address—one to access the pertinent link data in L_α, and one to fetch the word addressed in segment D. Realization of this minimum number of references implies use of the indirect addressing feature of the processor. Thus the link data for an established link will be an indirect word pair containing the generalized

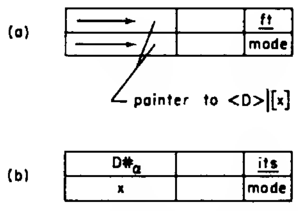


FIG. 11. States of the link data

address $D\#_\alpha|x$ (Figure 11a). Before the link is established, an attempt by a process of computation α to reference D|x through the link must lead to a trap of the process and transfer of control to the system routines that will establish the link and continue operation of the process. For this purpose a special form of indirect word pair is used which causes the desired trap. In Figure 11b this is indicated by the code *ft* in the addressing mode field of the pair. The segment number and word number fields of the indirect word can then be used to inform supervisory routines of the place to look to find the symbolic address $\langle D\rangle|[x]$ associated with the link. This address must be translated into a generalized address to establish the link. The operation of changing the link data to establish a link is called *linking.*

It is desirable to keep the procedure segment P self-contained if at all possible. Consequently the symbolic address $\langle D\rangle|[x]$ pointed to by the unestablished link should be part of the procedure segment P. Two look-up operations are required on the part of supervisory routines to establish the link. The symbolic reference name D must be associated with a specific segment through a search in the directory structure, and this segment must be made known to the process if a segment number has not already been assigned.

The word number corresponding to the symbolic word name x must also be determined. The set of associations between symbolic word names and word numbers for a segment is its *symbol table* and is part of the segment. Thus, in our example, a list of word numbers corresponding to symbolic word names that may appear in references to segment D from other segments is included as part of segment D at a standard position known to the system. This list is searched by a system routine to find the word number required to establish a link.

The Link Pointer. A remaining question is: How does a process produce the generalized address $L\#_\alpha|w$ required to access the link data? One might suppose that word address w could be fixed permanently at the time procedure segment P was created. This is not possible because the set of segments required by each process that might share use of procedure P will in general be unrelated: If the linkage sections of several procedures were placed in a single segment, assigning a fixed position to a link for all processes would produce intolerable conflicts. On the other hand, the code by which an intersegment reference is represented in segment P must be fixed and identical for all computations to meet the pure procedure constraint. Any data that allow different addresses to be formed from fixed code must reside in processor registers. By this argument we see the necessity of associating a *linkage pointer* with each process. The linkage pointer is a generalized address that resides in a dedicated base register (designated $\underline{\text{lp}}$). As shown in Figure 12, it is the origin $L\#_\alpha|s$ of the portion of a linkage segment that contains the links for intersegment references made from the segment being executed.

References to external segments are coded relative to the link pointer and have the form shown in Figure 12. The displacement k is determined by the coding of P and is invariant with respect to the process using P.

Procedure Call and Return. The coding used to transfer control to a subprocedure and the subsequent return of control must meet the requirements of programming generality. In particular, no assumptions may be made regarding the detailed coding of either the calling or called procedure other than those aspects uniformly established by convention. Conventions for four aspects of subroutine calling are relatively familiar:

(1) Transmission of arguments.

(2) Arranging for return of control.

(3) Saving and restoring processor state.

(4) Allocating private storage for the called procedure.

Item (4) is necessary in MULTICS because of the pure procedure requirement, and the generality requirement which forbids prior arrangement of a called procedure's storage needs. This private storage is supplied by associating the *stack segment* with each process in which a *frame* of private storage is reserved at each procedure call.

The frame is released upon return of control. This mechanism is implemented by the stack pointer (designated sp) which is the generalized address of the stack frame origin for the procedure in operation. The use of the stack segment makes every procedure in MULTICS automatically recursive by associating separate stack frames with successive entries into the same procedure. Due to the pure procedure requirement, only fixed arguments that do not depend on segment numbers may appear in procedure segments. Pointers and variable arguments must be placed in the stack segment, the linkage segment, or elsewhere. So that the language designer may have his choice of implementation, the argument pointer (designated ap) is at procedure entry the generalized address of the list of arguments for the called procedure.

In addition to these conventional requirements, the method of dynamic linking just described introduces one new problem: When process α, in executing procedure P, transfers control to procedure Q, the value of linkage

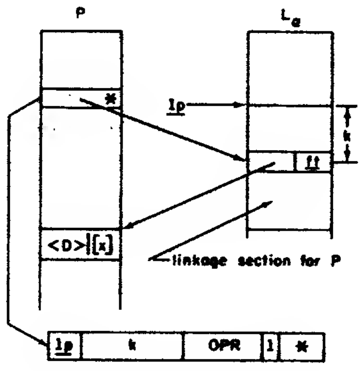


FIG. 12. Addressing the link data

pointer must be changed to the generalized address of the linkage section for procedure Q. Since the new value of the linkage pointer contains a segment number, it is private data of process α and cannot be placed in segment P or Q.

This problem requires a somewhat modified form of intersegment linkage from that used for data references. Since it is desirable that the machine code necessary to load the linkage pointer for a procedure segment be associated with that segment, the following solution was adopted. For each external entry point within a procedure segment, two additional instructions are placed in the procedure's linkage section at compilation time. The first instruction loads the linkage pointer with the appropriate value at procedure entry, and the second instruction transfers control to the entry point in the called procedure segment. Thus in establishing the link for an external procedure call, the generalized indirect address placed in the calling procedure's link data points to the corresponding instruction pair in the linkage section of the procedure being called. When control passes to the linkage segment during an external procedure call, the segment number portion of the desired linkage pointer is easily obtained from the procedure base register, since the process is now executing in the desired linkage segment.

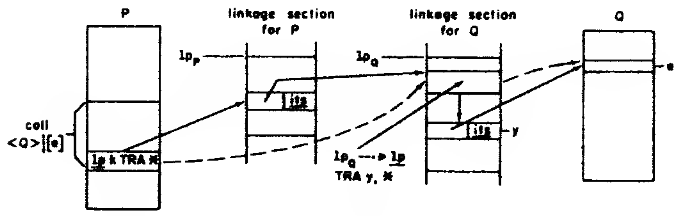


FIG. 13. Linkage mechanism for procedure entry

Figure 13 depicts the linkage mechanism required for an external procedure call from procedure P to segment Q at entry point e. The solid lines indicate the individual steps taken through indirect addresses, while the dashed lines indicate resulting flow of control.

In executing a call to an external procedure, the caller's machine conditions, including the procedure base register and program counter, are saved in the stack segment by the caller. Return from the called procedure can thus be effected by simply restoring the caller's machine conditions from the stack segment.

Acknowledgments. The evolution of the concepts presented in this paper represents the efforts of many members of the MULTICS programming staff. However, the authors wish to express particular appreciation of the work of F. J. Corbato and R. M. Graham in developing the basic design of the MULTICS linkage mechanism.

REFERENCES

1. CORBATO, F. J., AND VYSSOTSKY, V. A. Introduction and overview of the MULTICS system. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 185–197.
2. DALEY, R. C., AND NEUMANN, P. G. A general purpose file system for secondary storage. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part 1. Spartan Books, New York, pp. 213–229.
3. DENNIS, J. B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12*, 4 (Oct. 1965), 589–602.
4. ——, AND VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (Oct. 1966), 143–155.
5. DIJKSTRA, E. W. Cooperating sequential processes. Technological U., Eindhoven, The Netherlands.
6. GLASER, E. L., COULEUR, J. F., AND OLIVER, G. A. System design of a computer for time sharing applications. Proc. AFIPS 1965 Fall Joint Comput. Conf., Vol. 27, Part. 1. Spartan Books, New York, pp. 197–202.
7. GRAHAM, R. M. Protection in an information processing utility. *Comm. ACM 11*, 5 (May 1968), 365–369.
8. KILBURN, T., EDWARDS, D., LANIGAN, M., AND SUMNER, F. One level storage system. *IEEE Trans. EC-11*, 2 (April 1962), 223–235.
9. SALTZER, J. H. Traffic control in a multiplexed computer system. Tech. Rep. No. MAC-TR-30 (Ph.D. thesis). Project MAC, MIT, Cambridge, Mass., 1964.

Protection and the Control of Information Sharing in Multics

by J.H. Saltzer. Reprinted from ACM Fourth Symposium on Operating System Principles, Yorktown Heights, New York, October, 1973, with permission.

This paper provides a survey of all the different techniques, mechanisms, and design principles that underlie the control of access to information in Multics. Since it describes an area that is a subject of continuing research at M.I.T., its details (especially its list of weaknesses) ara going out of date quite rapidly. Nevertheless, the general concern of the Multics design that it support the need for privacy of individuals and organizations is best exhibited by a comprehensive snapshot of the mechanisms used.

PROTECTION AND CONTROL OF INFORMATION SHARING IN MULTICS

by

Jerome H. Saltzer

Massachusetts Institute of Technology
Department of Electrical Engineering and Project MAC

ABSTRACT

This paper describes the design of mechanisms to control sharing of information in the Multics system. Seven design principles help provide insight into the tradeoffs among different possible designs. The key mechanisms described include access control lists, hierarchical control of access specifications, identification and authentication of users, and primary memory protection. The paper ends with a discussion of several known weaknesses in the current protection mechanism design.

An essential part of a general-purpose computer utility system is a set of protection mechanisms which control the transfer of information among the users of the utility. The Multics system*, a prototype computer utility, serves as a useful case study of the protection mechanisms needed to permit controlled sharing of information in an on-line, general-purpose, information-storing system. This paper provides a survey of the various techniques currently used in Multics to provide controlled sharing, user authentication, inter-user isolation, supervisor-user protection, user-written proprietary programs, and control of special privileges.

Controlled sharing of information was a goal in the initial specifications of Multics[8, 11], and thus has influenced every stage of the system design, starting with the hardware modifications to the General Electric 635 computer which produced the original GE 645 base for Multics. As a result, information protection is more thoroughly integrated into the basic design of Multics than is the case for those commercial systems whose original specifications did not include comprehensive consideration of information protection.

Multics is an evolving system, so any case study must be a snapshot taken at some specific time. The time chosen for this snapshot is summer, 1973, at which time Multics is operating at M.I.T. using the Honeywell 6180 computer system. Rather than trying to document every detail of a changing environment, this paper concentrates on the protection strategy of Multics, with the goal of communicating those ideas which can be applied or adapted to other operating systems.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2095 which was monitored by ONR Contract No. N00014-70-A-0362-0006.

* A brief description of Multics, and a more complete bibliography, are given in the paper by Corbató, Saltzer, and Clingen[6].

What is new?

In trying to identify the ideas related to protection which were first introduced by Multics, a certain amount of confusion occurs. The design was initially laid out in 1964-1967, and ideas were borrowed from many sources and embellished, and new ideas were added. Since then, the system has been available for study to many other system designers, who have in turn borrowed and embellished upon the ideas they found in Multics while constructing their own systems. Thus some of the ideas reported here have already appeared in the literature. Of the ideas reported here, the following seem to be both novel and previously unreported:

- The notion of designing a comprehensive computer utility with information protection as a fundamental objective.
- Operation of the supervisor under the same hardware constraints as user programs, under descriptor control and in the same address space as the user.
- Facilities for user-constructed protected subsystems.
- An access control system applicable to batch as well as on-line jobs.
- Extensive human engineering of the user authentication (password) interface.
- Decentralization of administrative control of the protection mechanisms.
- Ability to allow or revoke access with immediate effect.

Multics is unique in the extent to which information protection has been permitted to influence the entire system design. By describing the range of protection ideas embedded in Multics, the extent of this influence should become apparent.

Design Principles

Before proceeding, it is useful to review several design principles which were used in the development of facilities for information protection in Multics. These design principles provided guidance in many decisions, although admittedly some of the principles were articulated only during the design, rather than in advance.

1. Every designer should know and understand the protection objectives of the system. At the present rather shaky stage of understanding of operating system engineering, there are many points at which an apparently "don't care" decision actually has a bearing on protection. Although these decisions will eventually come to light as the system design is integrated, a system design cannot withstand very many reversals of early design decisions if it is to be completed on a reasonable schedule and within a budget. By keeping all designers aware of the protection objectives, the early decisions are more likely to be made correctly.

2. Keep the design as simple and small as possible. This principle is stated so often that it becomes tiresome to hear. However, it bears repeating with respect to protection mechanisms, since there is a special problem: design and implementation errors which result in unwanted access paths will not be immediately noticed during routine use, since routine use usually does not include attempts to utilize improper access paths. Therefore, techniques such as complete, line-by-line auditing of the protection mechanisms are necessary; for such techniques to be successful, a small and simple design is essential.

3. Protection mechanisms should be based on permission rather than exclusion. This principle means that the default situation is lack of access, and the protection scheme provides selective permission for specific purposes. The alternative, in which mechanisms attempt to screen off sections of an otherwise open system, seems to present the wrong psychological base for secure system design. A conservative design must be based on arguments on why objects should be accessible, rather than on why they should not; in a large system some objects will be inadequately considered and a default of lack of access is more fail-safe. Along the same line of reasoning, a design or implementation mistake in a mechanism which gives explicit permission tends to fail by refusing permission, a safe situation, since it will be quickly detected. On the other hand a design or implementation mistake in a mechanism which explicitly excludes access tends to fail by not excluding access, a failure which may go unnoticed.

4. Every access to every object must be checked for authority. This principle, when applied methodically, is the primary underpinning of the protection system. It forces a system-wide view of access control which includes initialization, recovery, shutdown, and maintenance. It also implies that a foolproof method of identifying the source of every request must be devised. In a system designed to operate continuously, this principle requires that when access decisions are remembered for future use, careful consideration be given to how changes in authority are propagated into such local memories.

5. The design is not secret. The mechanisms do not depend on the ignorance of potential attackers, but rather on possession of specific, more easily protected, protection keys or passwords. This strong decoupling between protection mechanisms and protection keys permits the mechanisms to be reviewed and examined by as many competent authorities as possible, without concern that such review may itself compromise the safeguards. Peters[19] and Baran[2] discuss this point further.

6. The principle of least privilege. Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. If this principle is followed, the effect of accidents is reduced. Also, if a question related to misuse of a privilege occurs, the number of programs which must be audited is minimized. Put another way, if one has a mechanism available which can provide "firewalls", the principle of least privilege provides a rationale for where to install the firewalls.

7. Make sure that the design encourages correct behavior in the users, operators, and administrators of the system. Experience with systems which did not follow this principle revealed numerous examples in which users ignored or bypassed protection mechanisms for the sake of convenience. It is essential that the human interface be designed for naturalness, ease of use, and simplicity, so that users will routinely and automatically apply the protection mechanisms.

The application of these seven design principles will be evident in many of the specific mechanisms described in this paper.

Finally, in the design of Multics there were two additional functional objectives worth dwelling upon. The first of these was to provide the option of complete decentralization of the administration of protection specifications. If the system design forces all administrative decisions (e.g., protection specifications) to be set by a single administrator, that administrator quickly becomes a bottleneck and an impediment to effective use of the system, with the result that users begin adopting habits which bypass the administrator, often compromising protection in the bargain. Even if responsibility can be distributed among several administrators, the same effects may occur. Only by permitting the individual user some control of his own administrative environment can one insist that he take responsibility for his work. Of course, centralization of authority should be available as an option. It is easy to limit decentralization; it seems harder to adapt a centralized design to an environment in which decentralization is needed.

The second additional functional objective was to assume that some users will require protection schemes not anticipated in the original design. This objective requires that the system provide a complete set of handholds so that the user, without exercising special privileges, may construct a protection environment which can interpret access requests however he desires. The method used is to permit any user to construct a <u>protected subsystem</u>, which is a collection of programs and data with the property that the data may be accessed only by programs in the subsystem, and the programs may be entered only at designated entry points. A protected subsystem can thus be used to program any desired access control scheme.

The Storage System and Access Control Lists

The central fixture of Multics is an organized information storage system.[8] Since the storage system provides both reliability and protection from unauthorized information release, the user is thereby encouraged to make it the repository of all of his programs and data files. All use of information in the storage system is implemented by mapping the information into the virtual memory of some Multics process. Physical storage location is automatically determined by activity. As a result, the storage system is also used for all system data bases and tables, including those related to protection. The consequence of these observations is that one access control mechanism, that of the storage system, handles almost all of the protection responsibility in Multics.

Storage is logically organized in separately named data storage segments, each of which contains up to 262,144 36-bit words. A segment is the cataloguing unit of the storage system, and it is also the unit of separate protection. Associated with each segment is an access control list, an open-ended list of names of users who are permitted to reference the segment*. To understand the structure of the access control list, first consider that every access to a stored segment is actually made by a Multics process. Associated with each process is an unforgeable character string identifier, assigned to the process when it was created. In its simplest form, this identifier might consist of the personal name of the individual responsible for the actions of the process. (This responsible person is commonly called the principal, and the identifier the principal identifier.) Whenever the process attempts to access a segment or other object catalogued by the storage system, the principal identifier of the process is compared with those appearing on the access control list of the object; if any match is found access is granted.

Actually, Multics uses a more flexible scheme which facilitates granting access to groups of users, not all of whose members are known, and which may have dynamically varying membership. A principal identifier in Multics consists of several parts; each part of the identifier corresponds to an independent, exhaustive partition of all users into named groups. At present, the standard Multics principal identifier contains three parts, corresponding to three partitions:

1. The first partition places every individual user of the installation in a separate access control group by himself, and names the group with his personal name. (This partition is identical to the simple mechanism of the previous paragraph.)
2. The second partition places users in groups called projects, which are basically sets of users who cooperate in some activity such as constructing a compiler or updating an inventory file. One person may be a member of several projects, although at the beginning of any instance of his use of Multics he must decide under which project he is operating.
3. The third partition allows an individual user to create his own, named protection compartments. Private compartments are chiefly useful for the user who has borrowed a program which he has not audited, and wishes to insure that the borrowed program does not access certain of his own files. The user may designate which of his own partitions he wishes to use at the time he authenticates his identity*.

* The Multics access control list corresponds roughly to a column of Lampson's protection matrix. [16]

Although the precise description in terms of exhaustive partitions sounds formidable, in practice a relatively easy-to-use mechanism results. For example, the user named "Jones" working on the project named "Inventory" and designating the personal compartment named "a" would be assigned the principal identifier:

Jones.Inventory.a

Whenever his process attempts to access an object catalogued by the storage system, this three part principal identifier is first compared with successive entries of the access control list for the object. An access control list entry similarly has three parts, but with the additional convention that any or all of the parts may carry a special flag to indicate "don't care" for that particular partition. (We represent the special flag with an asterisk in the following examples.) Thus, the access control list entry

Jones.Inventory.a

would permit access to exactly the principal of our earlier example. The access control list entry

Jones.*.*

would permit access to Jones no matter what project he is operating under, and independent of his personally designated compartment. Finally, the access control list entry

.Inventory.

would permit access to all users of the "Inventory" project. Matching is on a part by part basis, so there is no confusion if there happens to be a project named "Jones".

Using multi-component principal identifiers it is straightforward to implement a variety of standard security mechanisms. For example, the military "need-to-know" list corresponds to a series of access control list entries with explicit user names but (possibly) asterisks in the remaining fields. The standard government security compartments are examples of additional partitions, and would be implemented by extending the principal identifier to four or more parts, each additional part corresponding to one compartment in use at a particular installation. (Every person would be either in or out of each such compartment.) A restriction of access to users who are simultaneously in two or more compartments is then easily expressed.

* The third partition has not yet been completely implemented. The current system uses the third partition only to distinguish between interactive and absentee use of the system.

We have used the term "object" to describe the entities catalogued by the storage system with the intent of implying that segments are not the only kinds of objects. Currently, four kinds of objects are implemented or envisioned:

1. Segments
2. Message queues (experimental implementation)
3. Directories (called catalogues in some systems)
4. Removable media descriptors (not yet implemented)

For each object, there are several separately controllable modes of access to the object. For example, a segment may be read, written, or executed as a procedure. If we use the letters r, w, and e for these three modes of access, an access control list entry for a segment may specify any of the combinations of access in table I. Certain access mode combinations are prohibited either because they make no sense (e.g., write only) or correct implementation requires more sophisticated machinery than implied by the simple mode settings. (For example, an execute-only mode, while appealing as a method for obtaining proprietary procedures, leaves unsolved certain problems of general proprietary procedures, such as protection of return points of calls to other procedures. The protection ring mechanism described later is used in Multics to implement proprietary procedures. The execute-only mode, while probably useful for less general cases, has not been pursued.)

Mode	Typical use
(none)	access denied
r	read-only data
re	pure procedure
rw	writeable data
rew	impure procedure

Table I: Acceptable combinations of access modes for a segment.

In a similar way, message queues permit separate control of enqueueing and dequeueing of messages, tape reel media descriptors permit separate control of reading, writing, and appending to the end of a tape reel, and directories permit separate control of listing of contents, modifying existing entries, and adding new entries. Control of these various forms of access to objects is provided by extending each access control list entry to include access mode indicators. Thus, the access control list entry

Smith.*.* rw

permits Smith to read and write the data segment associated with the entry.

It would have been simpler to associate an access mode with the object itself, rather than with each individual access control list entry, but the flexibility of allowing different users to have different access modes seems useful. It also makes possible exceptions to the granting of access to all members of a group. In the case where more than one access control list entry applies, with different access modes, the convention is made that the first access control list entry which matches the principal identifier of the requesting process is the one which applies. Thus, the pair of access control list entries:

Smith.Inventory.* (none)

.Inventory. rw

would deny access to Smith, while permitting all other members of the "Inventory" project to read and write the segment*. To insure that such control is effective, when an entry is added to an access control list, it is sorted into the list according to how specific the entry is by the following rule: all entries containing specific names in the first part are placed before those with "don't cares" in the first part. Each of those subgroups is then similarly ordered according to the second part, and so on. The purpose of this sorting is to allow very specific additions to an access control list to tend to take precedence over previously existing (perhaps by default) less specific entries, without requiring that the user master a language which permits him arbitrary ordering of entries. The result is that most common access control intentions are handled correctly automatically, and only unusually sophisticated intentions require careful analysis by the user to get them to come out right.

To minimize the explicit attention which s user must give to setting access control lists, every directory contains an "initial access control list". Whenever a new object is created in that directory, the contents of the initial access control list are copied into the access control list of the newly created object**. Only if the user wishes access to be handled differently than this does he have to take explicit action. Permission to modify a directory's contents implies also permission to modify its initial access control list.

The access control list mechanism illustrates an interesting subtlety. One might consider providing, as a convenience, checking of new access control list entries at the time they are made, for example to warn a user that he has just created an access control list entry for a non-existent person. Such checks were initially implemented in Multics,

* This feature violates design principle three, which proscribes selective exclusion from an otherwise open environment because of the risk of undetected errors. The feature has been provided nevertheless, because the alternative of listing every user except the few excluded seems clumsy.

** An earlier version of Multics did not copy the initial access control list, but instead considered it to be a common appendix to every access control list in that directory. That strategy made automatic sorting of access control list entries ineffective, so sorting was left to the user. As a result, the net effect of a single change to the common appendix could be different for every object in the directory, leading to frequent mistakes and confusion, in violation of the seventh design principle. Since in the protection area, it is essential that a user be able to easily understand the consequences of an action, this apparently more flexible design was abandoned in favor of the less flexible but more understandable one.

but it was quickly noticed that they represented a kind of compromise of privacy: by creating an access control list entry naming an individual, the presence or absence of an error message would tell whether or not that individual was a registered user of the system, thereby possibly compromising his privacy. For this reason, a name-encoding scheme which required checking of access control entry names at the time they were created was abandoned.

It is also interesting to compare the Multics access control scheme with that of the earlier CTSS system[6]. In CTSS, each file had a set of access restriction bits, applying to all users. Sharing of files was accomplished by permitting other users to place in their directories special entries called links, which named the original file, and typically contained further restrictions on allowable access modes. The CTSS scheme had several defects not present in the Multics arrangement:

1. Once a link was in place there was no way to remove it without modifying the borrower's directory. Thus, revocation of access was awkward.
2. A single user, using the same file via different links, could have different access privileges, depending on which link he used. Allowing access rights to depend on the name which happens to be used for an object certainly introduced an extra degree of flexibility, but this flexibility more often resulted in mistakes than in usefulness.
3. As part of a protection audit, one would like to be able to obtain a list of all users who can access a file. To construct that list, on CTSS, one had to search every directory in the system to make a list of links. Thus such an audit was expensive and also compromised other users' privacy.

Multics retains the concept of a link as a naming convenience, but the Multics link confers no access privileges -- it is only an indirect address.

Early in the design of Multics[8] an additional extension was proposed for an access control list entry: the "trap" extension, consisting of a one-bit flag and the name of a procedure. The idea was that for all users whose principal identifier matched with that entry, if the trap flag were on the procedure named in the trap extension should be called before access be granted. The procedure, supplied by the setter of the access control list entry, could supply arbitrary access constraints, such as permitting access only during certain hours or only after asking another logged in user for an OK. This idea, like that of the execute-only procedure, is appealing but requires an astonishing amount of supporting mechanism. The trap procedure cannot be run in the requesting user's addressing and protection environment, since he is in control of the environment and could easily subvert the trap procedure. Since the trap procedure is supplied by another user, it cannot be run in the supervisor's protection environment, either, so a separate, protected subsystem environment is called for. Since the current Multics protected subsystem scheme allows a subsystem to have access to all of its user's files, implementation of the trap extension could expose a user to unexpected threats from trap procedures on any data segment he touches. Therefore, at the least, a user should be able to request that he be denied access to objects protected by trap extensions, rather than be subject to unexpected threats from trap procedures. Finally, if such a trap occurs on every read or write reference to the segment, the cost would seem to be high. On the other hand, if the trap occurs only at the time the segment is mapped into a user's address space*, then design principle four, that every reference be validated, is violated; revocation of access becomes difficult especially if the system is operated continuously for long periods. The sum total of these considerations led to temporarily abandoning the idea of the trap extension, perhaps until such time as a more general domain scheme, such as that suggested by Schroeder[21] is available.

Both backup copying of segments (for reliability) and bulk input and output to printers, etc. are carried out by operator-controlled processes which are subject to access control just as are ordinary users. Thus a user can insure that printed copies of a segment are not accidentally made, by failing to provide an access control list entry which permits the printer process to read the segment**. Access control list entries permitting backup and bulk I/O are usually part of the default initial access control list. Bulk input of cards is accomplished by an operator process which reads them into a system directory, and leaves a note for the user in question to move them to his own directory. This strategy guarantees that there is no way in which one user can overwrite another user's segment by submitting a spurious card input request. These mechanisms are examples of the fourth design principle: every access to every object is checked for authority.

An administrative consequence of the access control list organization is that personal and project names, once assigned, cannot easily be reused, since the names may appear in access control lists. In principle, a system administrator could, when a user departs, unregister him and then examine every access control list of the storage system for instances of that name, and delete them. The system has been deliberately designed to discourage such a strategy, on the basis that a system administrator should not routinely paw through all the directories of all system users. Thus, the alternative scheme was adopted, requiring all user names, once registered, to be permanent.

Finally, the one most apparent limitation of the scheme as presently implemented is its "one-way" control of access. With the described access control list organization, the owner of a segment has complete control over who may access it. There are some cases in which users other than the owner may wish to see access restricted to an object which the owner has declared public. For example, an instructor of a class may for pedagogical purposes wish to require his students to write a

* Or, in traditional file systems, at the time the file is "opened".

** Of course, another user who has permission to read the segment could make a copy and then have the copy printed. Methods of constraining even users who have permission are the subject of continuing research[20].

particular program rather than make use of an equivalent one already publicly available in the system. Alternatively, a project administrator concerned about security may wish to insure that his project members cannot copy sensitive information into storage areas belonging to other users and which are not under his control. He may also want to prevent his project members from setting access control lists to permit access by users outside the project. This kind of control can be expressed in Multics currently only by going to the trouble of constructing a protected subsystem which examines all supervisor calls, thereby permitting complete control over which objects are mapped into the address space and what terms are added to access control lists. Fortunately, there have so far appeared only a few examples in which such control is required, and the escape suggested has proven adequate for those cases. A more general, yet quite simple, solution would be to associate with the user's process two constraining lists: a list of pathnames of directories whose contents he may access, and a list of access control list terms which he is permitted to place on access control lists. These two constraining lists would be set only by the project administrator or security officer. The constraining lists would be especially useful in the military security environment, since they would help in the construction of a list of items a defector might have had access to.

As is evident, the Multics access control list mechanism represents an engineering tradeoff among three conflicting goals: flexibility of expression, ease of understanding and use, and economy of implementation. Additional flexibility of expression was tried (e.g., the common access control list mechanism previously footnoted) with the conclusion that the additional confusion which results from accidental misuse of the generality can outweigh the benefits; apparently the correct direction is the opposite, toward simpler, less general, and more easily understandable protection structures.

<u>Hierarchical Control of Access Specifications</u>

Since in Multics every object, including a directory, must be catalogued in some directory, all objects are arranged into a single hierarchical tree of directories. This naming hierarchy also provides a hierarchy of control of access, through the ability to modify the contents of a directory. Since a directory entry consists of the name of some object and its access control list, having access to modify directory entries is interpreted to include the ability to modify the access control lists of all the objects catalogued in that directory. No further hierarchical control is provided; for example, there is no ability to say "Allow read access to Jones for all segments below this node in the naming tree". Such specifications are similar in nature to the "common access control list" mentioned before; they make it difficult for a user to be sure of all the consequences of a change to the access specification. For example, removing a specification such as that quoted above, which permits only reading, might render effective a forgotten access control term lower in the naming hierarchy which permits both reading and writing*.

* Early versions of Multics provided a limited form of higher-level specification in the form of ability to deny all use of a directory, and therefore of the objects contained within it. For the reasons suggested, this feature has been disabled.

Although it would appear that the hierarchical scheme provides an inordinate amount of power to a project administrator and, above him, to a system administrator, in practice it forces a careful consideration of the lines of authority over protected information, and explicit recognition of an authority hierarchy which already existed. In some environments, it would probably be appropriate to publicly log all modifications of directory access above some level, so as to provide a measure of control of the use of hierarchical authority. More elaborate controls might include requiring cooperative consent of some quasi-judicial committee of users for modification of high-level directory access. Such controls are relatively easy for an installation or a project to implement, using protected subsystems.

It is possible, by choosing access modes correctly, to use the hierarchical access control scheme in combination with the initial access control list to accomplish a totally centralized control of all access decisions. If, for example, a project administrator creates a directory for a user, places an initial access control list in that directory, and then grants to the new user permission only to add new entries to the directory, all such new entries would automatically receive a copy of the initial access control list determined by the administrator -- the user would have no control over who may use the objects he creates. By policy, a system administrator could run an entire installation under this tight control, and retain for himself complete authority to determine what access control list is placed on every object, as in IBM's Resource Security System[14]. Alternatively, any smaller portion of the naming hierarchy can be kept under absolute control by the person having authority to modify access control lists at the top node of the portion.

The other obvious alternative to a hierarchical control of modification of access control lists would be some form of self-control. That is, the ability to modify an access control list would be one of the modes of access controlled by the list itself. A very general version of this alternative has been explored by Rotenberg[20]. This alternative has not been tried out in the Multics context, partly because the implications of the hierarchical method were easier to understand in the first implementation. Probably the chief advantage of self-control of access modification would be that one could provide an individual a fully private work area in which no one -- manager, security officer, or system administrator -- could intrude. On the other hand, the implementation of a "locksmith" while easy to do may require introducing hidden access paths which are then subject to misuse*.

* A locksmith would be an administrator who can provide accountable intervention when mistakes are made. For example, if an organization's key data base is under the exclusive control of a manager who has been disabled in an automobile accident, the locksmith could then provide another manager with access to the file. It seems appropriate to formalize the concept of a locksmith so that appropriate audit trails and authority to be a locksmith

Also, one wonders how a self-control scheme would fit smoothly into an organization which does not usually give an individual the privilege of choosing his own office door lock. Clearly, the social and organizational consequences of the choice between these two design alternatives deserve further study.

Authentication of users

All of the machinery of access control lists, access modes, protected subsystems, and hierarchical control depend on an accurate principal identifier being associated with every process. Accuracy of identification depends on authentication of the user's claimed identity. A variety of mechanisms are used to help insure the security of this authentication. The general strategy chosen by Multics is to maintain individual accountability on a personal basis. Every user of a given installation (with one class of exception, noted later) is registered at the installation, which means that a unique name, usually his last name plus one or two initials, is permanently entered in a system registry. Associated with his name at the time he is registered is a password of up to eight ASCII characters. Whenever any person proposes to use the system, he supplies his unique name, at which point the system demands also that he provide his password.

Thus far, the authentication mechanism of Multics is essentially the same as for most other remote-accessed systems. However, Multics uses several extra measures related to user authentication, which are not often found in other systems. For one, all use of the system, whether interactive or absentee (batch) is authenticated interactively. That is, initiation of a batch job is not done on the basis of information found in a card reader. Arriving card decks are read in and held in on-line storage by a system process, for which an operator is responsible. All absentee jobs, whether they are to be controlled by files created from cards or files constructed interactively or files constructed by another program, must be initiated by some job already on the system, and whose legitimacy has been previously authenticated. Although a chain of absentee job requests can be developed, the chain must have begun with an interactive job, which requires interactive authentication. In the simplest case, the individual responsible goes to an interactive console, identifies and authenticates himself, and requests execution of the job represented by the incoming card deck. If necessary, the request will automatically wait until the card deck arrives, so that the user need not wait for the operator or for a card reader queue*. Thus, no job is every run without prior positive identification of the responsible party. Note that for installations in which responsibility for card controlled jobs is considered unimportant, it is rather trivial to construct a Multics program, run under the responsibility of the card reader operator, which accepts and runs as a job anything found in the card reader. All such jobs would be run in processes bearing the principal identifier of the card reader operator, and are thus constrained in the range of on-line information which they can access. The inviolate principle of access control remains that on-line authentication of identity, by presenting a password, is required in order to start a process labeled with a particular desired principal identifier. Note also that the fact that a job happens to be operated without an interactive terminal has no bearing on its privileges, except as explicitly controlled by its principal identifier. Finally, to handle the situation where a busy researcher asks a friend to submit the batch job, a proxy login scheme permits the friend to identify himself, under his own password, and then request that the job be run under the principal identifier of the original researcher. The system will permit proxy logins only if the person responsible for the principal identifier to be used has previously authorized such logins by giving a list of proxies*.

As to protection of passwords, several facilities are provided. The user may, after authenticating himself, change his password at any time he feels that the old one may have been compromised. A program is available which will generate a new random eight-character password with English digraph statistics, thereby making it pronounceable and easy to memorize, and minimizing the need for written copies of the password. Users are encouraged to obtain their passwords from this program, rather than choosing passwords themselves, since human-chosen passwords are often surprisingly easy to guess. Passwords are stored in the file system in mildly encrypted form, using a one-way encryption scheme along the lines suggested by Wilkes[29]. As a result, passwords are not routinely known by any system administrator or project administrators, and there is never any occasion for which it is even appropriate to print out lists of passwords. If, through some accident, a stored password is exposed, its usefulness is reduced by its encrypted form.

When the user is requested to give his password, at login time, the printer on his terminal is turned off, if possible, or else a background of garbling characters is first printed in the area where he is to type his password. Although the user could be indoctrinated to tear off and destroy the piece of paper containing his password, by routinely protecting it for him the system encourages a concern for security on the part of the user. In addition, if the user's boss (or someone from four levels of management higher) happens to be looking over his shoulder as he logs in, the user is not faced with the awkward social problem of scrambling to conceal his password from a superior who could potentially take offense at an implication that he is not to be trusted with the information.

A time-out is provided to help protect the user who leaves his terminal, is distracted, and forgets to log out. If no activity occurs for a period, a logout is automatically generated. The length of the time-out period can be adjusted to suit the needs of a particular installation. Similarly, whenever service is interrupted by a system failure for more than a moment, a new login

---

can be well-defined. The alternative of sending a system programmer into the computer room with instructions to directly patch the system or its data may leave no audit trail and almost certainly encourages sloppy practice.

* The automatic wait is not yet implemented.

* The proxy login is not yet implemented.

is required of all interactive users, since some users may have given up and left their terminals.

Finally, several logging and penetration detection techniques help prevent attacks via the password routine. If a user provides an incorrect password, the event of an incorrect login attempt is noted in a threat-monitoring log, and the user is permitted to try again, up to a limit of ten times at which point the telephone (or network) connection is forcibly broken by the system, introducing delay to frustrate systematic penetration attempts*. Whenever a user logs in, the time and physical location (terminal identification) of his previous login are printed out in his greeting message, thus giving him an opportunity to notice if his password has been used by someone else in his absence. Similarly, monthly accounting reports break down usage by shift and services used, and may be reviewed on-line at any time, thereby providing an opportunity for the individual to compare his pattern of use with that observed by the system, and perhaps to thereby detect unauthorized use. If either of these mechanisms suggests unauthorized use, the individual involved may ask the system administrator to check the system log, which contains an entry for every login and logout giving date and time, terminal type used, and terminal identification, if any.

For a project which maintains especially sensitive information, the project administrator may designate the initial procedure to be executed by some or all processes created using the name of that project as part of its principal identifier. This initial procedure, supplied by the project administrator, has complete control of the process, and can demand further authentication (e.g., a one-time password or a challenge-response scheme,) perform project logging of the result, constrain the user to a subset of the available facilities, or initiate a logout sequence, thereby refusing access to the user. In the other direction, some projects may wish to allow unlimited public access to their files. If so, the project administrator may indicate that his project will accept login of unauthenticated users. In such a case, the system does not demand a password, instead assigning the personal name "anonymous" to the principal identifier of the process involved, using the name of the responsible project for the second part of the principal identifier. The principal identifier "anonymous" is the one exception to the registration scheme mentioned earlier. Allowing anonymous users does not compromise the security of the storage system, since the principal identifier is constrained, and all storage system access is based on the principal identifier. The primary use of anonymous users has been for educational purposes, in which all students in a class are to perform some assignment. Sometimes, this feature is coupled with the project-designated initial procedure, so that the project may implement its own password scheme, or control what facilities are made available, so as to limit its financial liability. Some statistical analysis and data-base development projects also permit anonymous use of data-retrieval programs.

The objective of many of these mechanisms, such as simple registration of every user, the proxy login, the anonymous user, concealment of printed passwords, and user changeable passwords, together with a storage system which permits all authorized sharing of information, is to provide an environment in which there is never any need for anyone to know a password other than his own. Experience with the earlier CTSS system demonstrated that by omitting any of these features, the system itself may encourage borrowing of passwords, with an attendent reduction in overall security.

Primary Memory Protection

We may consider the access control list to be the first level of mechanism providing protection for stored information. Most of the burden of keeping users' programs from interfering with one another, with protected subsystems, and with the supervisor is actually carried by a second level of mechanism, which is descriptor-based. This second level is introduced essentially for speed, so that arbitration of access may occur on every reference to memory. As a result, the second level is implemented mostly in hardware in the central processing unit of the Honeywell 6180. Of course, this strategy requires that the second level of mechanism be operated in such a way as to carry out the intent expressed in the first level access control lists.

As described by Bensoussan et al.[4] the Multics virtual memory is segmented to permit sharing of objects in the virtual memory, and to simplify address space management for the programmer. The implementation of segmentation uses addressing descriptors, a technique used, for example, in the Burroughs B5000 computer systems[9]. The Burroughs implementation of a descriptor is exclusively as an addressing and type-labeling mechanism, with protection provided on the basis that a process may access only those objects for which it has names. In Multics, the function of the descriptor* is extended to include modes of access (read, write, and execute) and to provide for protected subsystems which share object names with their users. Evans and LeClerc[10] were among the first to describe the usefulness of such an extension.

* With ASCII passwords chosen to match English digraph frequency, a little less than four bits of information are represented by each character (despite the eight or nine bits required to store the characters.) An eight character password thus carries about 30 bits of information, which would require about 10^9 guesses using an information theoretic optimum guessing strategy. If one mounted a simultaneous attack from 100 computer-driven terminals, and the system-imposed delays average only 10 milliseconds per attempt, about 10^5 seconds, or one full day of systematic attack would be required to guess a password. Although use of a uniformly random password generator would increase this work factor by several orders of magnitude, resistance to use of hard-to-remember passwords and the need to make written copies might act to wipe out the gain. Of course, this work factor calculation presumes that the attacker has no further basis on which to narrow the range of password possibilities, for example, by knowing that the user in question may have chosen his own password, or by wiretapping a previous login.

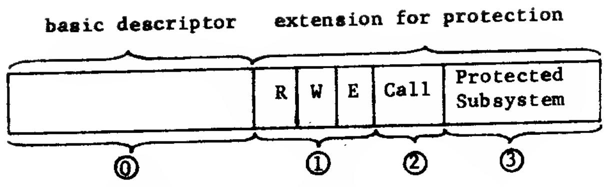
* With the exception of type identification, which is <u>not</u> provided in Multics.

As shown in figure one, there are three classes of descriptor extensions for protection purposes: mode control, protected subsystem entry control, and control on which protected subsystems may use the descriptor at all. Every reference of the processor to the segment described by this descriptor is thus checked for validity.

The virtual address space of a Multics process is implemented with an array of descriptors, called a descriptor segment, as in figure two. Every reference to the virtual memory specifies both a segment number (which is interpreted as an index into the descriptor segment) and a word number within the segment.

Figure two also helps illustrate why the protection information is associated with the addressing descriptor rather than with the data itself*. Each computation is carried out in its own address space, so each computation has its own private descriptor segment. Using this mechanism, a single physical segment may appear in different address spaces with different access privileges for different users, even though they are referring to the same physical data. Since in a multiprocessor system such as Multics two such processes may be executing simultaneously, a single protection specification associated with the data is not sufficient. Having the protection specification associated with the descriptor allows for such controlled sharing to be handled easily.

* The alternate option is chosen, for example, in the IBM 360/67 and the IBM 370 "Advanced Function" virtual memory systems[24].



⓪ Physical address and size of the segment based on this descriptor.

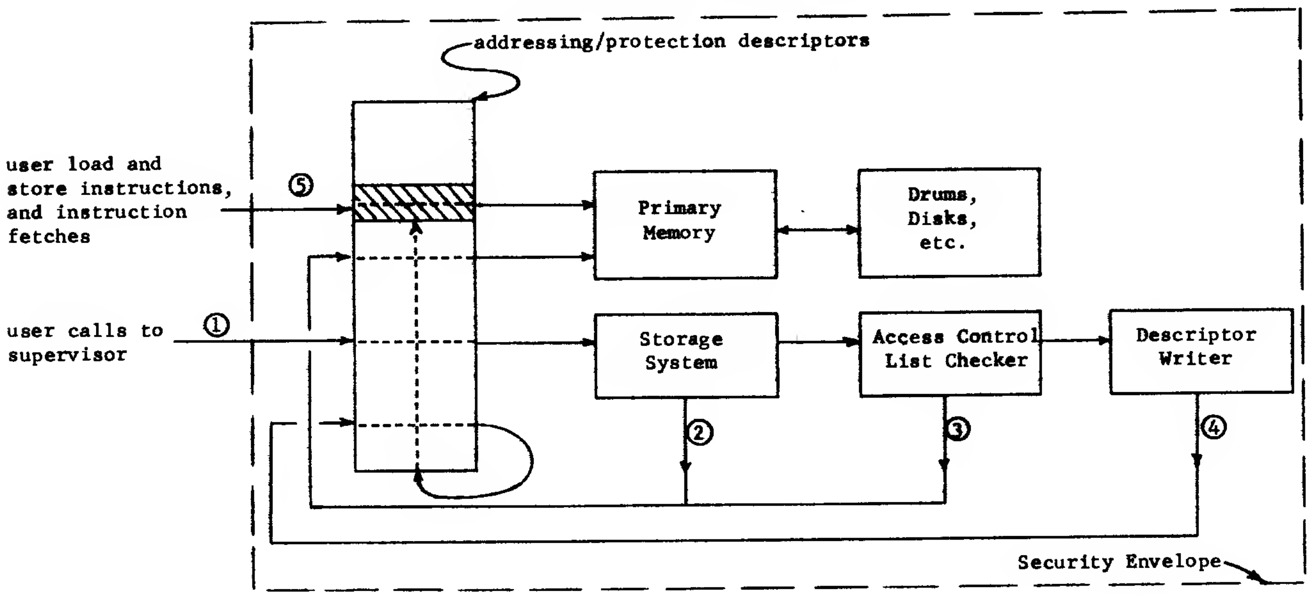
① Bits separately controlling permission to read, write, and execute the contents of the segment based on this descriptor.

② Control of permission to enter a protected subsystem which has entry points in the segment based on this descriptor.

③ Controls on which (hierarchically arranged) protected subsystems may use this descriptor.

Figure 1 -- A Multics descriptor.

An unusual feature of the descriptors used in Multics is embodied in the second and third extensions of figure one. Together, they allow hardware enforcement of protected subsystems. A protected subsystem is a collection of procedures and data bases which are intended to be used only by calls to designated entry points, known in Multics


addressing/protection descriptors
user load and store instructions, and instruction fetches
user calls to supervisor
⑤
①
Primary Memory
Drums, Disks, etc.
Storage System
Access Control List Checker
Descriptor Writer
②
③
④
Security Envelope

① Call to storage system to add object to virtual memory.

② VM access by storage system to locate object in directory structure. (Includes recursive invocation of storage system to add directories to VM).

③ VM access by access control list checker to read principal identifier and access control list.

④ VM access to write new addressing and protection descriptor into descriptor segment.

⑤ Caller accesses new object.

Figure 2 -- Descriptor management in Multics. The Multics supervisor is treated as a protected subsystem.

as gates. If this intention is hardware enforced, it is possible to construct proprietary programs which cannot be read, data base managers which return only statistics rather than raw data to some callers, and debugging tools which cannot be accidentally disabled. The descriptor extensions are used to authenticate subroutine calls to protected subsystems. Two important advantages flow from using a hardware checked call:

1. Calls to protected subsystems use the same structural mechanisms as do calls to unprotected subroutines, with the same cost in execution time. Thus a programmer does not need to take the fact that he is calling a protected subsystem into account when he tries to estimate the performance of a new program design.
2. It is quite easy to extend to the user the ability to write protected subsystems of his own. Without any special privileges, any user may develop his own proprietary program, data-screening system, or extra authentication system, and be assured that even though he permits others to use his protected subsystem, the information he is protecting receives the same kind of security as does the supervisor itself.

In support of call protection, hardware is also provided to automatically check the addresses of all arguments as they are used, to be sure that the caller has access to them. Checking the range of the argument values is left to the protected subsystem.

Protected subsystems are formed by using the third field of the descriptor extension of figure one. To simplify protected subsystem implementation, Multics imposes a hierarchical constraint on all subsystems which operate within a single process: each subsystem is assigned a number, between 0 and 7, and it is permitted to use all of those descriptors containing protected subsystem numbers greater than or equal to its own. Among the descriptors available to a subsystem may be some permitting it to call to the entry points of other protected subsystems. This scheme goes by the name rings of protection, and is more completely described by Graham[12] and by Schroeder and Saltzer[22].* As far as is known, the only previously existing systems to permit general, user-constructed protected subsystems are the M.I.T. PDP-1 time-sharing system[1] and the CAL operating system[15].

The descriptor-based strategy permits two further simplifying steps to be taken:

1. All information in the storage system is read and written by mapping it into the virtual memory, and then using load and store instructions whose validity is checked by the descriptor mechanism.
2. The supervisor itself is treated as an example of a protected subsystem, which operates in a virtual memory arbitrated by descriptors, exactly the same as do the user programs which it supports.

* A more general approach, not yet implemented, but which removes the restriction that the protected subsystem be hierarchical, is described by Schroeder in his doctoral thesis[21].

The reasons why the first step provides simplification for the user have been discussed extensively in the literature[4,13]. The second step deserves some more comment. By placing the supervisor itself under the control of the descriptors, as in figure two, a rather substantial benefit is achieved: the supervisor then operates with the same addressing and machine language code generation environment as the user, which means that supervisor programs may be constructed using the same compilers and debugging tools available to a user. The effect on protection is non-trivial: programs constructed and checked out with more powerful tools tend to have fewer errors, and errors in the supervisor which compromise protection often escape notice.

Perhaps equally important is that the determination of whether one is in or out of the supervisor is not based on some processor mode bit which can be accidentally left in the wrong state when control is passed to a user program. Instead, the addressing privileges of the current protected subsystem are governed by the subsystem identification, located in the descriptor of the segment which supplied the most recent instruction. Every transfer of control to a different program is thus guaranteed to automatically produce addressing privileges appropriate to the new program. If a supervisor procedure should accidentally transfer to a location in a user procedure, that procedure will find that the protection environment has automatically returned to the state appropriate for running user procedures.

Finally, the descriptors are adjusted to provide only the amount of access required by the supervisor, in consonance with design principle six. For example, procedures are not writeable, and data bases are not executable. As a result, programming errors related to using incorrect addresses tend to be immediately detected as protection violations, and do not persist into delivered systems. If one reviews the operation of Multics starting with the initial loading of the system on an empty machine, he will find that only the first hundred or so instructions do not use descriptors. Once a descriptor segment has been fashioned, all memory references by the processor from that point on are arbitrated by descriptors.

These mechanisms do not prohibit the supervisor from making full use of the hardware when appropriate. Rather, they protect against accidental overuse of supervisor privileges. Clearly, the supervisor must be able to write into the descriptor segment, in order to initially set it up, and also to honor requests to map additional objects of the storage system into segments of the virtual memory. This adjustment of descriptors is done with great care, using a single procedure whose only function is to construct descriptors which correspond to access control list entries. A call to the storage system which results in adjustment of a descriptor is illustrated in figure two. In this figure, it is worth noting that even the writing of the descriptor is done with use of a descriptor for the descriptor itself. Thus there is little danger of accidentally modifying a descriptor segment belonging to some other user, since the only descriptor segment routinely appearing in the virtual memory of this process is its own.

Entries to the supervisor which implement "special privileges" (e.g., the operator may have the privilege of shutting the system down) are generally controlled by ordinary access control lists, either on the gates of supervisor entries, or in some cases by having the supervisor procedure access some data segment before proceeding with the privileged operation. If the user attempting to invoke the privilege does not appear on the access control list of the data segment, an access violation fault will occur, rather than an unauthorized use of the privilege.

The final step of "locking up" the supervisor lies in management of source-sink input-output. Recall first that all access to on-line catalogued information of the storage system is handled by direct mapping into the virtual memory. Thus, input and output operations in Multics consist only of true source-sink operations, that is of streams of information which enter or leave the system. Such operations are performed by hardware I/O channels, following channel programs constructed by the I/O system in response to I/O requests of the calling program. These I/O channel programs are placed in a part of the virtual memory accessible only to the supervisor*. Similarly, all input data is read into a protected buffer area, accessible only to the supervisor. Only after the input has arrived and the supervisor has had a chance to check it is it turned over to the user, either by copying it, or by modifying a descriptor to make it accessible to the user. A similar, inverse pattern is used on output. Since during I/O neither the data nor the channel program is accessible to the user, there is no hesitation about permitting him to continue his computation in parallel with the I/O operation. Thus, fully asynchronous operations are possible.

The system is initialized from a magnetic tape which contains copies of every program residing in the most protected area. In this way, the integrity of the protection mechanisms depends on protecting only one magnetic tape, and is independent of the contents of the secondary storage system (disk and drums) which are more exposed to compromise by maintenance staff. On the other hand, since the system is designed for continuous operation, there appears to be no need for a separate package consisting of passwords and clearance information as suggested by Weissman[28].

To round out the discussion of primary and virtual memory protection, we should consider storage residues. A storage residue is the data copy left in a physical storage device after the previous user has finished with it. Storage residues must be carefully controlled to avoid accidental release of information. In a virtual memory system, the only way a storage residue could be examined would be to read from a previously unused part of the virtual memory. By convention, in Multics, the supervisor provides pages of zeros in response to such attempts. Since all access to on-line storage is via the virtual memory, no additional mechanism is required to insure that a user never sees a residue from the storage system.

<u>Weaknesses of the Multics Protection Mechanisms</u>

One is always hesitant to list the weaknesses in his system, for a variety of reasons. Often, they represent mistakes or errors of judgement, which are embarrassing to admit. Such a list provides an easy target for detractors of a design, and in the protection area provides an invitation for potential attackers at production installations which happen to be using the system. In the case of a system still evolving, such as Multics, known weaknesses are being corrected as rapidly as feasible, so any list of weaknesses is rapidly obsolete. And finally, any list of weaknesses is almost certainly incomplete, being subject to all of the built-in blindnesses of its authors. Nevertheless, such a list is quite useful, both to look for specific interesting unsolved problems, and also to establish what level of considerations are still considered relevant by the designers of the system. The weaknesses described here begin with two major areas, followed by several smaller problems.

Probably the most important weakness in the current Multics design lies in the large number of different program modules which have the ability, in principle, to compromise the protection system. Of the 2000 program modules which comprise Multics, some 400, or 20%, are in the "most protected area", consisting of system initialization, the storage system, miscellaneous supervisor functions, and system shutdown. Although all of these 400 modules operate using the descriptor-based virtual memory described earlier, the descriptors serve for them only as protection against accidentally generated illegal address references; these modules are not constrained by the inability to construct suitable descriptors in the same way as the remaining 1600 modules and user programs. Thus any of these 400 modules (averaging perhaps 200 lines of source code each) might contain an error which compromises the security mechanisms, or even a security violation intentionally inserted by a system programmer. The large number of programs and the very high internal intricacy level frustrates line by line auditing for errors, misimplementation, or intentially planted trapdoors. This weakness is not surprising for the first implementation of a sophisticated system, and upon review it is now apparent that with mild software restructuring plus help from specialized hardware the number of lines of code in the most protected area can be greatly reduced --

---

* And to the I/O channels, which use absolute addresses. If separate I/O channels were available to each physical device and the I/O channels used the addressing descriptors, protected supervisor procedures would not be required for I/O operations after device assignment (which requires a descriptor to be constructed.)

Here is an example of a place where building a new system, rather than modifying an old one, has simplified matters. On some computer systems, the user constructs his own channel programs, and may even expect to modify them dynamically during channel operation. It is quite hard to invent a satisfactory scheme for protecting other users against such I/O operations without placing restrictions on their scope, or inhibiting parallel operation of the user with his I/O channel programs.

perhaps by as much as an order of magnitude. In examining many specific examples, there seem to have been three common, interrelated reasons for the extra bulk currently found in the protected area:

- economics: at the time of design, a function could be implemented more cheaply in the most protected region. Since the protection ring mechanism was originally simulated by software, there were design decisions based on the assumption that calls across ring boundaries were expensive.
- rush to get on the air: in the hurry to get an initial version of the system going, a shortcut was found, which required unnecessarily placing a module in the most protected region.
- lack of understanding: a complex subsystem waa not carefully enough analyzed to separate the parts requiring protection; the entire subsystem was therefore protected.

With hardware-supported protection rings, hindsight, and the experience of a complete working implementation, it is apparent that a smaller "most protected area" can be constructed. It now appears possible to make complete auditing a feasible task. A project is now underway to test this hypothesis by attempting to develop an auditable version of the most protected region of Multics.

The second serious weakness in the current Multica design is in the complexity of the user interface. In creating a new segment, a user should specify permitted lists of users and projects, specify allowed modes of access for each, decide whether or not backup copies should be allowed and whether or not bulk I/O should be permitted for the segment, and whether or not the segment should be part of a protected subsystem. He should check that permissions he has given to modify higher-level directories interact in the desired way with his current intent. A variety of defaults have been devised to reduce the number of explicit choices which need be made in common cases: as already mentioned, a per-directory "initial access control list" is by default assigned to any new segment created in that directory. The defaults merely hide the complex underlying structure, however, and do not help the user with an unusual protection requirement, who must figure out for himself how to accomplish his intentions amid a myriad of possiblities, not all of which he understands. The situation for a project administrator, who can control the initial program his users get, and may perhaps force all of his users to interact via a limited, protected subaystem is similar, but with fewer defaults and more possibilities available.

The solution to thia problem lies in better understanding the nature of the typical user's mental description of protection intent, and then devising interfaces which permit more direct specification of that protection intent. As an example, a graduate student devised a simple Multics program which prints a liat of all users who may force access to a segment (by virtue of having modify access to some higher level directory.) This list does not correspond to any single access control list found anywhere in the system, yet it is clearly relevant to one's image of how the segment is protected. Setting up the mechanisms of access control lists, accessibility modes, and rings of protection perhaps should be viewed as a problem of programming in which, as usual, the structures available in initial designs do not correspond directly with the user's way of thinking, even though there may be some way of programming the structure to accomplish any intent. In the area of protection, the problem haa a special edge, aince if a user, through confusion, devises an overly permissive protection specification, he may not discover his mistake until too late.

At a level of significance well below the two major points of system size and user interface complexity are several other kinds of problems. These problems are felt to be less significant not becauae they cannot be exploited as easily, but rather because the changes required to strengthen these areas are straightforward and relatively easy to implement. These problems include:

1. Communication links are weak. Of course, any use of switched telephone lines leads to vulnerability, but provision for integration of a Lucifer-like system[23] for end-to-end encryption of messages sent over public lines or through a communication network would probably be a desirable (and simple) addition. As an example of a typical problem in this area, the Bell System 202C6 DATAPHONE dataset, which is used for 1200 bps terminals, does not include provision for reporting telephone line disconnection to the computer system during data output transmission. If a user accidentally hangs up his telephone line during output, another user dialing to the same port on the computer may receive the output, and capture control of the process. Although remedial measures such as requiring reauthentication every few minutes could be used, automatic detection of the line disconnection would be far more reassuring. (Note that for the more commonly used 103A DATAPHONE dataset, which does report telephone line disconnections, this problem does not exist; upon observing the dropping of the carrier detect line from the dataset, Multics immediately logs the user out.)
2. The operator interface is weak. The primary interface of the operator is as a logged-in user, where his interactions can be logged, verified, and suitably restricted. However, he has a secondary interface: the switches and lights of the hardware itself. It would appear that the potential for error or sabotage via this route is far higher than necessary. If every hardware switch in the system were both readable and settable by (protected supervisor) programs, then all such switches could be declared off limits to the operator, and perhaps placed behind locked panels. Since all operator interaction would then be forced to take place via his terminal, his requests can be checked for plausibility by a program. What has really gone wrong here is a failure to completely reconsider the role of the operator in a computer system operating as a utility. Functions such as operation of card readers and printers do not require access to switches on the side of the processor -- or even physical presence in the same room as the computer, for that matter. The decision that a system failure has occurred and the

appropriate level of recovery action to take are probably the operator functions which are hardest to automate or decouple from the physical machine room, but certainly much movement in this direction would be easy to accomplish.

3. Users are permitted to specify their own passwords, leading to easy-to-guess passwords. The resulting loss of security has already been well documented in the literature[25], and this method has been used at least once to improperly obtain access to Multics at M.I.T., when a programmer chose as his Multics password the same password he used on another, unsecured time-sharing system. A better strategy here would be to force the use of system-generated randomly chosen passwords, and also to place an expiration date on them, to force periodic password changes. For sensitive applications, or situations where the password must be exposed to unknown observers (as in using a system via the ARPA network), the system should provide lists of one-time passwords.

4. The supervisor interface is vulnerable to mis-implementation. Although this difficulty could be described as a specific example of a supervisor too large and complex to audit, it is worth identifying in its own right. The problem has to do with checking the range of arguments passed to the supervisor. The hardware automatically checks that argument addresses are legitimately accessible to the caller, and completely checks all use of pointer variables as indirect addresses. However, it provides no help in determining whether the ultimate argument values are "reasonable" for the supervisor entry in question. Each entry must be prepared to operate correctly (or at least safely) no matter what combination of argument values is supplied by the caller. Certain kinds of interfaces make for difficulty in auditing a program to see if it properly checks range of arguments. For example, if the allowed range of one argument depends on the result of computation which is based in part on another argument, then it may be hard to enforce a programming standard which requires that all supervisor entries check the range of all their arguments before performing any other computation. The current Multics interface has examples of situations in which, to verify that a supervisor entry is correctly programmed so that it does not blow up when presented with an illegal argument, one must trace hundreds of lines of code and many subroutine calls. Such interfaces discourage routine auditing of the supervisor interface, and probably result in some undetected implementation errors. It would be interesting to explore the design of argument range-checking hardware, which would force the system programmer to declare the allowed range of arguments for his entries, and thereby force out into the open the existence of arguments whose range is not trivially testable, for interface design revision.

5. Secondary storage residues are not cleared until they are reassigned. When a segment is deleted, all descriptors for the physical storage area are destroyed, and the area is marked as reusable. No further descriptors for the storage area will ever be constructed without first clearing the storage area, but meanwhile the residue remains intact. In principle, there is no way to exploit these residues using the system itself, but automatic overwriting of the residues at the time of deletion would provide an additional safeguard against accidents, and guarantee that a segment, once deleted, is not accessible even to a hardware maintenance engineer. A similar problem exists for the magnetic tapes containing backup copies of segments. In at least one case on another time-sharing system, the persistence of backup copies has proved embarrassing: a government agency requested that a file containing a list of special telephone access codes be completely deleted; the installation administrator found himself with no convenient way to purge the residues on the backup tapes. These tapes should probably be encrypted, using per-segment keys known only by the operating system. It is an interesting problem to construct a strategy for safely encrypting backup copy tapes, while ensuring that encrypting keys do not get destroyed upon system failure, making the backup copies worthless.

6. Over-privileged system administrator. Some system functions have been organized in such a way that the administrators of the system require more privilege than really necessary. For example, measures of secondary storage usage are stored in the using directory rather than in an account file. As a result, the administrative accounting programs which prepare bills for secondary storage use must have access to read every directory in the storage system. For another example, the "locksmith" function, mentioned earlier, is currently implemented by giving the locksmith permission to modify the root directory of the storage system directory hierarchy. Thus the locksmith has the unaudited ability to grant himself access to every file in the storage system. Such a design means that one of the easiest ways to attack is to attempt to influence the system administrator, possibly by surreptitiously inserting traps in some program he is likely to use* while running a process whose principal identifier needlessly permits extensive privileges. The counter measure, currently partially implemented, is to provide administrators with protected subsystems from which they cannot escape, which are certified to exercise a minimum of privilege, and which maintain audit trails.

7. Ponderous backup copy and retrieval scheme. It has been noticed that the general method currently used for indexing the contents of storage system backup copy tapes is weak, so that the only effective way to identify a desired copy of a damaged segment is to permit the user to manually scan printed journals of the names of the segments copied onto each tape. These journals contain the names of

* This technique has been described as the "Trojan Horse" attack[5].

other users' segments and directories, and were intended for use only for emergency situations and with proper clearance. Unfortunately, the number of retrieval requests which can be handled on other than an emergency basis is a sensitive function of the quality of the tools available for searching the journals automatically while maintaining privacy. A simple scheme based on a protected subsystem for searching journals has recently been proposed, but is not yet implemented.

8. Counter-intelligence techniques have not been exploited. Although logs of suspicious events (such as incorrectly supplied passwords) are maintained no true counter-intelligence strategies are employed. For example, Turn, et al. [26] have suggested inserting carefnlly monitored apparent flaws in the system. These flaws would be intended to attract a would-be attacker; any attempt to exploit them would result in an early warning of attack and an opportunity to apprehend the attacker.

9. Some areas of potentially vulnerability have not been examined. These include vulnerability to undetected failures of the hardware protection apparatus[17],* electromagnetic radiation from the physical hardware machine[3], and traffic analysis possibilities, using performance measurement tools available to any user.

It is interesting to note that none of these nine specific weaknesses represent intrinsic difficulties of full-scale computer utility systems -- relatively straightforward modification can easily strengthen any of these areas. In fact, neither the two major weaknesses nor the nine specific ones represent "holes" in the sense of being immediately exploitable by an attacker. Rather, they are areas in which an attacker is more likely to discover a method of entry cansed by misimplementation, misunderstanding, or mismanagement of an otherwise securable system. Thus we might describe the protection system as usable, though with known areas of weakness.

Conclusions

This paper has surveyed the complete range of information protection techniques which have been applied to a specific example of a system designed for production use as a computer utility. Over three years of experience in a production environment at M.I.T. has demonstrated that the mechanisms are generally useful. A commonly asked question (especially in the light of recent experiences with attempts to add security to other commercially available computer systems) is "how much performance is lost?" This question is difficult to answer since, as is evident, the protection structure is deeply integrated into the system and cannot be simply "turned off" for an experiment.* However, one significant observation may be made. In general, the protection mechanisms are closely related to naming mechanisms, and can be implemented with a minimum of extra fuss in a system which provides a highly structured naming environment. Thus, the users of Multics apparently have found that the overall package of a structured virtual memory with protection comes at an acceptable price.

The Multics protection mechanisms were designed to be basic and extendable, rather than a complete implementation of some specialized security model. Thus there are mechanisms which may be used to provide the multilevel security classification (top secret, secret, confidential, unclassified) and the access compartments of the U.S. governmental security system[27]. If one wished to precisely imitate the government security system, he could do so without altering the operating system. In this sense, Multics differs with, say, SDC's ADEPT[28] and IBM's Resource Security System[14], both of which specifically implement models of the government security system, but which do not permit, for example, user-written program-protected data bases.

We should also note that the Multics system was designed to be <u>securable</u>, which is different than stating that any particular site is actually operated in a completely secured fashion. Such matters as machine room security, certification of of hardware maintenance engineers and system operators, and telephone wire tapping are largely outside of the scope of operating system design. In addition, correct administration can be encouraged by the design of an operating system, but not enforced. Further we have reported the design of the system, realizing that its implementation has not yet been completely audited and therefore may contain trivial programming errors which affect protection.

Acknowledgements

As is usual in any large system design, many individuals have contributed ideas and suggestions, and a complete acknowledgement is very hard to compose. Professor E.L. Glaser provided the firm conviction that information protection was a reasonable goal during the critical initial design period of the Multics system. He also suggested several of the design principles and many of the specific protection mechanisms which were ultimately included. Professor R.M. Graham worked out the initial design of the protection ring mechanism, and Professor M.D. Schroeder expanded that design to include automatic argument validation and complete hardware support. Integration of protection into the storage system was accomplished by R.C. Daley. More recent upgradings of the user interface have been designed by V.L. Voydock, R.J. Feiertag, and T.H. VanVleck. P.A. Belmont,

* Although the 6180 hardware is less vulnerable than some. An asynchronous processor-memory interface tends to stop when an error occurs rather than proceeding with wrong data; complete instruction decoding explicitly traps all but legal operation codes and addressing modifiers; and the multiprocessor organization helps obviate the need for pipelines and other accident-prone highly-tuned logic tricks.

* In analogy, we may consider a mouse. The mouse has an elaborate system which maintains a constant body temperature, where, for example, a lizard does not. There is a sense in which the mouse is thereby less efficient, but one may also credibly argue that the question of efficiency is incorrectly posed. In a similar way, comparison of systems with and without protection may also be incorrect. (Analogy thanks to Carla M. Vogt.)

D.A. Stone, and M.A. Meer developed an early internal memorandum which helped articulate the design issues. Others offering significant help include Professor F.J. Corbató, C.T. Clingen, D.D. Clark, M.A. Padlipsky, and P.G. Neumann. Of course, every system programmer who worked in the most protected region of Multics has also contributed by his extra care and understanding of the protection objective.

References

1. Ackerman, W.B., and W.W. Plummer, "An Implementation of a Multiprocessing Computer System," ACM Symposium on Operating Systems Principles, October, 1967, Gatlinburg, Tennessee.
2. Baran, P., "Security, Secrecy, and Tamper-Free Considerations," On Distributed Communications 9, Rand Corp. Technical Report RM-3765-PR.
3. Beardsley, C.W., "Is your computer insecure?" IEEE Spectrum 9, 1 (January, 1972), pp. 67-78.
4. Bensoussan, A., C.T. Clingen, and R.C. Daley, "The Multics Virtual Memory: Concepts and Design," Comm. ACM 15, 5 (May, 1972), pp. 308-318.
5. Branstad, D.K., "Privacy and Protection in Operating Systems," Computer 6, 1, 1973, pp. 43-47.
6. The Compatible Time-Sharing System: A Programmer's Guide, M.I.T. Press, 1966.
7. Corbató, F.J., J.H. Saltzer, and C.T. Clingen, "Multics: The First Seven Years," AFIPS Conf. Proc. 40, (1972 SJCC), pp. 571-583.
8. Daley, R.C., and P.G. Neumann, "A General-Purpose File System for Secondary Storage," AFIPS Conf. Proc. 27, (1965 FJCC), pp. 213-229.
9. The Descriptor -- A Definition of the B5000 Information Processing System, Burroughs Corporation, Business Machines Group, Sales Technical Services, Systems Documentation, Detroit, Michigan, 1961.
10. Evans, D.C., and J.Y. LeClerc, "Address Mapping and the Control of Access in an Interactive Computer," AFIPS Conf. Proc. 30, (1967 SJCC), pp. 23-30.
11. Glaser, E.L., "A Brief Description of Privacy Measures in the Multics Operating System," AFIPS Conf. Proc. 30, (1967 SJCC), pp. 303-304.
12. Graham, R.M., "Protection in an Information Processing Utility," Comm. ACM 11, 5 (May, 1968), pp. 365-369.
13. Holland, S.A., and C.J. Purcell, "The CDC Star-100 -- A Large Scale Network Oriented Computer System," IEEE International Computer Society Conf., (September, 1971), pp. 55-56.
14. IBM Application Program Manual "OS/MVT with Resource Security, General Information and Planning Manual," File no. GH20-1058-0, IBM Corporation, December, 1971.
15. Lampson, B.W., "An Overview of the CAL Time-Sharing System," Computer Center, University of California, Berkeley, (September 5, 1969).
16. Lampson, B.W., "Protection," Proc. 5th Princeton Conf. on Information Sciences and Systems, (March, 1971), pp. 437-443.
17. Molho, L.M., "Hardware aspects of secure computing," AFIPS Conf. Proc. 36, (1970 SJCC) pp. 135-141.
18. Needham, R.M., "Protection Systems and Protection Implementations," AFIPS Conf. Proc. 41, Vol I, (1972 FJCC), pp. 572-578.
19. Peters, B., "Security considerations in a multi-programmed computer system," AFIPS Conf. Proc. 30, (1967 SJCC), pp. 283-286.
20. Rotenberg, L., "Making Computers Keep Secrets," Ph.D. Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, September, 1973. (Also available as M.I.T. Project MAC Technical Report TR-116.)
21. Schroeder, M.D., "Cooperation of Mutually Suspicious Subsystems in a Computer Utility," Ph.D. Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, September, 1972. (Also available as M.I.T. Project MAC Technical Report TR-104.)
22. Schroeder, M.D., and J.H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," Comm. ACM 15, 3 (March, 1972), pp. 157-170.
23. Smith, J.L., W.A. Notz, and P.R. Osseck, "An Experimental Application of Cryptography to a Remotely Accessed Data System," Proc. ACM 1972 Conf., pp. 282-297.
24. System 370 Principles of Operation, IBM Systems Reference Library File no. GA22-7000.
25. "Third Party ID Aided Program Theft," ComputerWorld V, 14, April 7, 1971.
26. Turn, R., R. Fredrickson, and D. Hollingworth, Data Security at the Rand Corporation, Rand Corp. Technical Report P-4914, October, 1972.
27. Ware, W., et al., "Security Controls for Computer Systems, Rand Corp. Technical Report R-609, 1970. (Classified Confidential).
28. Weissman, C., "Security Controls in the ADEPT-50 Time-Sharing System," AFIPS Conf. Proc. 35, (1969 FJCC), pp. 119-133.
29. Wilkes, M.V., Time-Sharing Computer Systems, American Elsevier Publishing Co., 1968.

A Hardware Architecture for Implementing Protection Rings

by M.D. Schroeder and J.H. Saltzer. Reprinted from Communications of the ACM, 15, 3, March, 1972, pp. 157-170, with permission. Copyright 1972 by the Association for Computing Machinery.

The casual reader may wish to explore only the first half dozen pages of this paper, which describes in full detail the rather unusual hardware protection mechanism in use in the current Multics system. As far as is known, Multics and the CAL operating system (developed at the University of California at Berkeley) are the only two systems thus far developed which permit construction of general, user-constructed, protected subsystems. This paper describes the mechanisms which make this feature possible in Multics. Since the paper is recent, the terminology and description are generally up-to-date. The mechanisms described here are exactly the ones implemented on the Honeywell 6180 computer system.

A Hardware Architecture for Implementing Protection Rings

Michael D. Schroeder and Jerome H. Saltzer
Massachusetts Institute of Technology*

Protection of computations and information is an important aspect of a computer utility. In a system which uses segmentation as a memory addressing scheme, protection can be achieved in part by associating concentric rings of decreasing access privilege with a computation. This paper describes hardware processor mechanisms for implementing these rings of protection. The mechanisms allow cross-ring calls and subsequent returns to occur without trapping to the supervisor. Automatic hardware validation of references across ring boundaries is also performed. Thus, a call by a user procedure to a protected subsystem (including the the supervisor) is identical to a call to a companion user procedure. The mechanisms of passing and referencing arguments are the same in both cases as well.

Key Words and Phrases: protection, protection rings, protection hardware, access control, hardware access control, computer utility, time-sharing, shared information, segmentation, virtual memory, Multics

CR Categories: 4.32, 6.21

Introduction

The topic of this paper is the control of access to stored information in a computer utility. The paper describes a set of processor access control mechanisms that were devised as part of the second iteration of the hardware base for the Multics system. These mechanisms provide a hardware implementation of protection rings which limit the access privileges of an executing program.

Multics is a general purpose, multiple user, interactive computer system developed at Project MAC of MIT in a joint effort with the Cambridge Information Systems Laboratory of Honeywell Information Systems Inc. and, until 1969, the Bell Telephone Laboratories. It was built and is being run as an experiment in designing, implementing, operating, and evaluating a prototype computer utility. (Reference [14] contains a bibliography of publications on Multics.)

Multics is currently implemented on a Honeywell 645 computer system. The 645 represents a first attempt to define a suitable hardware base for a computer utility. While containing special logic to support a segmented virtual memory, the 645 processor [10] provides only a limited set of access control mechanisms, forcing software intervention to implement protection rings. In the course of Multics development a second iteration of the design of the hardware base has been undertaken. The resulting new hardware system is being built as a re-

* Project MAC and Department of Electrical Engineering, 545 Technology Square, Cambridge, MA 02139. Work reported herein was supported in part by Project MAC, an MIT research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N00014-70-A-0362-0001.

Presented at the Third ACM Symposium on Operating Systems Principles, Palo Alto, California, October 18–20, 1971.

placement for the 645 using the technology of the Honeywell 6000 series computer systems. The new processor includes an improved set of access control mechanisms, described here, which implement rings almost completely in hardware. These mechanisms were developed from a scheme described in [16]. Although specifically designed for Multics, the mechanisms are applicable to any computer system which uses segmentation as a memory addressing scheme.

This paper begins by establishing the general need to control access to stored information in a computer utility and by presenting several criteria for comparing different sets of access control mechanisms. Relevant aspects of the organization of segmented memories are then sketched, and the processor mechanisms for implementing protection rings are described. The paper concludes by illustrating how rings can be used and by evaluating the impact of a hardware system design.

Access Control in a Computer Utility

Protection of computations and information is an important aspect of a computer utility. The multiple users of a computer utility have different goals and are responsible to different authorities. Such a diverse group will use the same system only if it is possible for them to achieve independence from one another. On the other hand, a great potential benefit of a computer utility is its ability to allow users to easily communicate, cooperate, and build upon one another's work. The role of protection in a computer utility is to control user interaction—guaranteeing total user separation when desired, allowing unrestricted user cooperation when desired, and providing as many intermediate degrees of control as will be useful.

While there are many manifestations of protection in a computer utility, most may be related to controlling access to stored information. Because stored information represents both data and executable procedure, control of access to stored information serves to regulate information processing as well.

Four criteria can be applied to a set of access control mechanisms to judge its usefulness in a computer utility: functional capability, economy, simplicity, and programming generality. The first means that a set of access control mechanisms should be able to meet an interesting set of user protection needs in a natural way. The ability to meet interesting protection needs must be a quality of the basic mechanisms, while the ability to do so in a natural way is a quality of their user interface. An obvious goal in designing new protection mechanisms is to maximize functional capability.

The second criterion, economy, means that the cost of specifying and enforcing a particular kind of access constraint with a set of mechanisms should be so low that it is not an important consideration in determining the type of access control to be used in a particular application. In addition, cost should be proportional to the functional capability actually used. The existence of access control mechanisms with sophisticated capabilities should cost no extra to those with unsophisticated needs. Cost includes the subsystem complexity and user inconvenience that result from use of the access control mechanisms, as well as any associated extra storage space and execution time.

Simplicity is the third criterion. While it is true that simplicity often leads to economy, something more is at stake. For a set of access control mechanisms to be accepted there must be confidence that no way exists to circumvent it. The best way to achieve confidence is to keep the mechanisms so simple that they may be completely understood. With respect to access control mechanisms, lack of simplicity often implies lack of security.

The fourth criterion, programming generality, is often neglected. It means that individual procedures may be combined easily into larger units without understanding or altering their internal organizations. Programming generality allows sharing to be effective in encouraging users to build upon one another's work. An implication of programming generality of relevance to access control mechanisms is that it should be possible to change the protection environment of procedures and collections of procedures without altering their internal structure.

It clearly is difficult to design access control mechanisms which satisfy all four of these criteria simultaneously. Increases in functional capability come at the expense of economy, simplicity, and programming generality. The challenge in designing a set of access control mechanisms is to maximize functional capability within the constraints of the other three criteria. In the following sections a set of hardware access control mechanisms that was devised in the course of Multics development is described. These mechanisms appear to provide a significant improvement in the simultaneous satisfaction of the four criteria as compared with the mechanisms in the initial Multics implementation.

Segmented Virtual Memory Environment

The processor access control mechanisms described here regulate the ability of an executing program to reference information in a segmented virtual memory. As a basis for understanding these access control mechanisms this section briefly reviews the structure of a typical segmented virtual memory. (See [1–3] for detailed descriptions of several segmented virtual memories.)

A machine language program for a segmented environment does not reference memory by absolute address. Rather, its memory consists of independent segments identified by number. Each segment is a separate array of words. A two-part address (s, w) identifies word w of the segment numbered s.

The collection of segments in the virtual memory is defined by a descriptor segment containing an array of segment descriptor words (SDW's). Each SDW can describe a single segment in the virtual memory. The number of a segment is just the index of the corresponding SDW in the descriptor segment. Among other things, an SDW contains the absolute address of the beginning of the corresponding segment in memory. The absolute address of the beginning of the descriptor segment is contained in the descriptor base register (DBR) of a processor. Each processor contains logic for automatically translating two-part addresses into the corresponding absolute addresses. Address translation, done with an indexed retrieval of the appropriate SDW from the descriptor segment, occurs each time a word in the virtual memory is referenced, i.e. each time an instruction, indirect word, or instruction operand reference is made by an executing program.

Storage for segments is usually allocated with a paging scheme in scattered fixed-length blocks. If used, paging is also taken into account by the address translation logic, but is totally transparent to an executing machine language program. Paging, if appropriately implemented, need not affect access control; it will be ignored in the remainder of this paper.

Changing the absolute address in the DBR of a processor will cause the address translation logic to interpret two-part addresses relative to a different descriptor segment. This facility can be used to provide each user of the system with a separate virtual memory. A single segment may be part of several virtual memories at the same time, allowing straightforward sharing of segments among users.

Controlling Access in a Segmented Virtual Memory

To provide a framework for discussion, three specific assumptions true of Multics are introduced. First, a process with a new virtual memory is created for each user when he logs in to the system, and the name of the user is associated with the process. The process is the active agent of the user, and is his only means of referencing and manipulating information stored on-line. Second, on-line storage is organized as a collection of segments of information. A process can reference a segment of on-line storage only if the segment is first added to the virtual memory of the process. Third, the users that are permitted to access each segment are named by an access control list associated with each segment. As will be seen, any system providing access control of the type under discussion will probably have analogous assumptions. The application of the rest of the discussion to other systems with segmented virtual memories is straightforward.

Adding a segment to a virtual memory, an operation performed by supervisor programs, provides the initial opportunity for controlling access to information stored on-line. The name of the user associated with a process must match some entry on the access control list of a segment before the supervisor will add that segment to the the virtual memory of the process.

Once a segment is included in the virtual memory, however, finer control on access is required. (If a process could, say, write in any segment to which it had access, little sharing of information among users would occur.) If this finer control is to be effective against arbitrary machine language programs constructed by users, it must be implemented as hardware access validation on each reference. The structure of the virtual memory makes it natural to record these finer constraints in the SDW associated with each segment. Since the processor must examine the SDW for a segment each time that segment is referenced by two-part address anyway, there is little effort added to validate the intended access against constraints recorded there. With this structure it is also possible to change the allowed access to a segment by changing the finer constraints recorded in the SDW, and to expect the change to be immediately effective, although the need for such dynamic changes is rare.

Flags which enable a segment to be read, written, and executed are natural constraints to record in each SDW. The value for each flag comes from the access control list entry which matched the name of the user associated with the process. An attempt by a process to change the contents of a word of a segment, for example, would be allowed by the processor only if the write flag were *on* in the SDW for the segment. This mechanism provides individual control on the ability of each user's process to read, write, and execute the words in each segment stored on-line. It also makes a segment the smallest unit of information that can be separately protected.

With the access control mechanisms described so far, all programs executed as part of some process have the same information accessing capabilities. However, there seems to be an intrinsic need in many computations for the access capabilities of a process to vary as the execution point passes through the various programs that direct the computation. The most obvious examples of this need are explicit invocations of supervisor programs during the course of a computation. The execution point may pass from a user program to a supervisor program to initiate an input/output operation or change the access control list of a segment, and then pass back to the user program. Presumably the executing supervisor program can access information in some way that the user program cannot. In a system that allows and encourages sharing of information among users, other examples appear. For instance, user A may wish to allow user B to access a sensitive data segment, but only through a special program, provided by A, that audits references to the segment. During the course of a computation in a process of user B, access to the sensitive data segment should be allowed only when the execution point is in the special program provided by A.

The word "domain" is frequently associated with a set of access capabilities. The examples above point to an intrinsic need for multiple domains to be associated with a process and for the domain in which the process is executing to occasionally change as the execution point passes from one program to another. A descriptor segment with read, write, and execute flags in the SDW's defines a single domain. Additional mechanisms are required to allow multiple domains to be associated with a single process.

A very general set of access control mechanisms would place no restriction on the number of domains which could be associated with a process, and would force no restrictive relationships to exist among the sets of access capabilities included in the domains. Unfortunately, devising such a set of access control mechanisms that also meets the criteria of economy, simplicity, and programming generality is a difficult research problem. (See [5, 7, 8, 12, 13, 17] for several approaches that have been explored.) In Multics the strategy was adopted of limiting the number of domains which may be associated with a process, and of forcing certain relationships to exist among the sets of access capabilities included in the domains. The result is protection rings.

The characterization of rings as a restricted implementation of domains is the result of hindsight. When developed, rings were viewed as a natural generalization of the supervisor/user modes that provided protection in many computers. This path of development was chosen because it solved the most pressing problems of access control involved in the prototype computer utility and, due to the inherent simplicity of the idea, it was a path that the Multics designers felt confident they could successfully complete. Even today rings appear to provide an effective trade-off among the criteria mentioned above.

Protection Rings

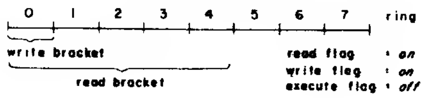
Associated with each process are a fixed number of domains called protection rings. These r rings are named by the integers 0 through $r - 1$. The access capabilities included in ring m are constrained to be a subset of those in ring n whenever $m > n$. Put another way, the sets of access capabilities represented by the various rings of a process form a collection of nested subsets, with ring 0 the largest set and ring $r - 1$ the smallest set in the collection. Thus, a process has the greatest access privilege when executing in ring 0, and the least access privilege when executing in ring $r - 1$. The total ordering of the sets of access capabilities defined by the consecutively numbered rings of a process is the property which allows a straightforward implementation of rings in hardware.

As described earlier, the permission flags for each segment in the virtual memory of a process simply indicate that the segment can or cannot be read, written, or executed by the process. With the addition of rings, the flags must be extended to indicate which rings include each access capability. Because of the nested subset property of rings, the capability, say, to write a particular segment, if available to a process at all, is included in all rings numbered less than or equal to some value w. The range of rings over which this write permission applies is called the write bracket of the segment for the process. Read and execute brackets for each segment can be established in the same way. A process is permitted to read, write, or execute a segment in its virtual memory only if the ring of execution of the process is within the proper bracket.

A partial hardware implementation of rings places numbers indicating the top of each bracket of a segment in the SDW of the segment, along with the read, write, and execute flags. If a flag is *on*, then the number specifies the extent of the corresponding bracket. Turning a flag *off* indicates that the corresponding access capability is not included in any ring of the process. For example, a data segment might have its execute flag turned *off* or a pure procedure segment might have its write flag turned *off*. A register is added to the processor to record the current ring of execution of the process. The processor can then validate each reference to a segment by making the obvious comparisons when the SDW for the segment is examined for address translation.

Figure 1 illustrates the flags and brackets that might be associated with a writable data segment for some process. (In Multics, eight was chosen as the appropriate number of rings. Eight rings are shown in the examples, although more or fewer rings might be appropriate in another system.)

Fig. 1. Example access indicators for a writable data segment.



The association of multiple domains of protection with a process generates the need for a new kind of access capability—the capability to change the domain of execution of a process. Since changing the domain of execution has the potential to make additional access capabilities available to a process, it is an operation that must be carefully controlled. An understanding of the sort of control required can be gained by reviewing the purpose of domains. A domain provides the means to protect procedure and data segments from other procedures that are part of the same computation. Using domains, it should be possible to make certain access capabilities available to a process only when particular programs are being executed. Restricting the start of execution in a particular domain to certain program locations, called gates, provides this ability, for it gives the program sections that begin at those locations complete control over the use made of the access capabilities included in the domain. Thus, changing the domain of execution must be restricted to occur only as the result of a transfer of control to one of these gate locations of another domain.

With a completely general implementation of domains, each domain could provide protection against the procedures executing in all other domains of a process. The corresponding property of rings is that the protection provided by a given ring of a process is effective against procedures executing in higher numbered rings. Switching the ring of execution to a lower number makes additional access capabilities available to a process, while switching the ring to a higher number reduces the available access capabilities. Thus, the downward ring switching capability must be coupled to a transfer of control to a gate into the lower numbered ring. Gates are specified by associating a (possibly empty) list of gate locations with each segment in the virtual memory of a process. If the execution point of the process is transferred to a segment while the ring of execution is above the top of the execute bracket for the segment, then the transfer must be directed to one of the gate locations in the segment. If the transfer is to a gate, then the ring of execution of the process will switch down to the top of the execute bracket of the segment as the transfer occurs. If the transfer is not directed to one of the gate locations, then the transfer is not allowed.

To provide control of this downward ring switching capability which is consistent with the subset property of rings, a gate extension to the execute bracket of a segment is defined. The gate extension specifies the consecutively numbered rings above the execute bracket of the segment that include the "transfer to a gate and change ring" capability for the segment. The gate list and the gate extension to the execute bracket can both be specified with additional fields in each SDW.

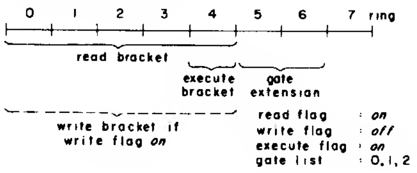
In contrast to downward ring changes, switching the ring of execution to a higher-numbered ring can only decrease the available access capabilities of a process. Thus, an upward ring switch is an unrestricted operation that can be performed by any executing procedure. (The instruction to be executed immediately following an upward ring switch must come from a segment that is executable in the new, higher-numbered ring.) For programming convenience, the upward ring switch may be coupled to a special transfer instruction.

The abstract description of rings is now one step from completion. The last step comes from the observation that for each procedure segment in the virtual memory of each process there is a lowest-numbered ring in which that procedure is intended to execute. In order to provide the means for preventing the accidental transfer to and execution of a procedure in a ring lower than intended, the requirement that execute brackets have a lower limit at ring 0 is relaxed and instead an arbitrary lower limit is allowed. For many procedure segments the execute bracket will include exactly one ring—the ring in which the procedure is intended to execute. Procedure segments with wider execute brackets normally will contain commonly used library subroutines that are certified as acceptable for execution in any of several rings.

The arbitrary lower limit on the execute bracket of a segment can be implemented by using the field of an SDW which specifies the top of the write bracket to specify the bottom of the execute bracket as well. The double use of this field does not appear to remove any interesting functional capability. In fact, it eliminates an unwanted degree of freedom in access specification, thereby removing the potential to make certain types of errors, such as allowing both writing and execution of a segment in more than one ring of a process.

Figure 2 shows example access indicators for a pure procedure segment containing gates, and illustrates how the execute and write brackets specified in an SDW must be related.

Fig. 2. Example access indicators for a pure procedure segment which contains gates.



The gate list and the numbers specifying the read, write, and execute brackets and gate extension in each SDW all come from the access control list entry which permitted the process to include the corresponding segment in its virtual memory, as did the values for the read, write, and execute flags.

Call and Return

As argued above, a change in the domain of execution of a process can occur only when the executing procedure transfers control to a gate of another domain. In the context of most programming languages, an interprocedure transfer represents a subroutine call, a return following a call, or a nonlocal goto. Linguistically, all three operations produce a change in the environment of the execution point; this change affects the binding of variable names to virtual storage locations. The call operation has the additional function of transmitting arguments and recording a return point. Performing these functions generally requires the cooperation of both the procedure initiating the operation and the procedure receiving control. If a call, return, or goto changes the domain of execution because it happens to be directed to a gate location of another domain, then the situation becomes more complicated, for neither procedure can depend upon the other to cooperate. An important simplification introduced by restricting domains to a ring structure is that a procedure may assume the cooperation of procedures in lower-numbered rings.

When procedures are shared among different processes and different domains, the addressing environment is usually defined via processor registers, for the procedures must be pure and it is not convenient to embed addresses within them. Part of the function of the call, return, and goto operations is to properly update this environment pointer. In Multics, pure procedures are used with a per process stack, and a stack pointer register provides the required environment definition. The stack of a process is implemented with a separate segment for each ring being used. The stack segment for procedures executing in ring *n* has read and write brackets that end at ring *n*. Thus, stack areas for these procedures are not accessible to procedures executing in any ring $m > n$. In the following discussion the stack pointer register is used as a typical example of the required environment pointer.

The most common ways of changing the ring of execution of a process are a call to a gate of a lower-numbered ring and the subsequent upward return. A downward call represents the invocation of a user-provided protected subsystem or a supervisor procedure. Because the Honeywell 645 was designed around the usual supervisor/user protection method, the version of Multics for this machine implements rings by trapping to a supervisor procedure when downward calls and upward returns are performed. The hardware mechanisms detailed in the next section eliminate the need to trap in these cases. Using these improved hardware access control mechanisms, downward calls and upward returns occur without the intervention of a supervisor procedure and are performed by the same object code sequences that perform all calls and returns.

It is the nested subset property of rings that makes a straightforward hardware implementation of downward calls and upward returns possible. Because of this property, the called procedure automatically has all access capabilities required to reference any arguments that the calling procedure can legitimately specify and to return to the calling procedure in the ring from which it called. However, three problems remain. First, the called procedure must have a way of finding a new stack area without depending upon information provided by the calling procedure. Second, the called procedure must have a way of validating references to arguments, so that it cannot be tricked into reading or writing an argument that the caller could not also read or write. Finally, the called procedure must have a way of knowing for certain the ring in which the calling procedure was executing, so that the called procedure cannot be tricked into returning control to a ring not as high as that of the calling procedure.

The key to solving the first problem, finding a new stack area, is a rule relating the segment number of the stack segment for a ring to the ring number. Using this rule, the processor automatically calculates the segment number of the proper stack segment for the called procedure's ring of execution. By convention, a fixed word of each stack segment can point to the beginning of the next available stack area. Thus, the stack segment number alone can provide the called procedure with enough information from which to construct its own stack pointer. Because the processor provides the stack segment number, no procedure executing in a higher-numbered ring, e.g. the calling procedure, can affect the value of the stack pointer for the called procedure.

The second problem, validating argument references, is solved by providing processor mechanisms which allow a procedure to assume the more restricted access capabilities of any higher-numbered ring for particular operand references. Using these mechanisms, the called procedure can validate access when referencing arguments as though execution were occurring in the (higher-numbered) ring of the calling procedure. Thus, the called procedure, even though it is executing in a ring with more access capabilities than the ring of the calling procedure, can prevent itself from reading or writing any argument that the calling procedure could not also read or write.

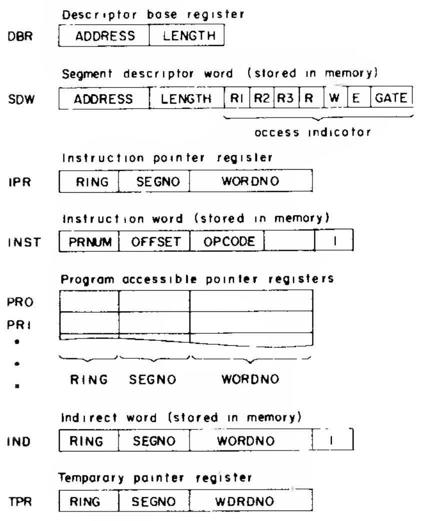
The final problem, knowing the ring of the caller, is solved by having the processor leave in a program accessible register the number of the ring in which execution was occurring before the downward call was made. The subsequent return is made to that ring. Thus the calling procedure has no opportunity to lower the number of the ring to which the return is made.

The next two sections describe in more detail how downward calls, argument referencing and validation, and upward returns are implemented. Before proceeding to that description, however, there are two other possibilities to consider: a call and return that do not change the ring of execution, and an upward call and the subsequent downward return. The first presents no protection problem, as both the calling and the called procedures have available the same set of access capabilities. The hardware mechanisms for downward calls and upward returns also work when no change of ring is needed.

The last possibility is more difficult to handle. An upward call occurs when a procedure executing in ring *n* calls an entry point in another procedure segment whose execute bracket bottom is $m > n$. When the call occurs, the ring of execution will change to *m*. The subsequent return is downward, resetting the ring of execution to *n*. These cases exhibit two unpleasant characteristics of a general cross-domain call and return that were not present in the other cases.

The first is that the calling procedure may specify arguments that cannot be referenced from the ring of the called procedure. (For a downward call, the nested subset property of rings guaranteed that this could not happen.) There are at least three possible solutions to this problem. One is to require that the calling procedure

Fig. 3. Schematic description of relevant storage formats and processor registers.


Descriptor base register
DBR
ADDRESS
LENGTH
Segment descriptor word (stored in memory)
SDW
ADDRESS
LENGTH
R1
R2
R3
R
W
E
GATE
access indicator
Instruction pointer register
IPR
RING
SEGNO
WORDNO
Instruction word (stored in memory)
INST
PRNUM
OFFSET
OPCODE
I
Program accessible pointer registers
PR0
PR1
RING
SEGNO
WORDNO
Indirect word (stored in memory)
IND
RING
SEGNO
WORDNO
I
Temporary pointer register
TPR
RING
SEGNO
WORDNO

specify only arguments that are accessible in the higher-numbered ring of the called procedure. This solution compromises programming generality by forcing the calling procedure to take special precautions in the case of an upward call. Another possible solution is to dynamically include in the ring of the called procedure the capabilities to reference the arguments. Because a segment is the smallest unit of information for which access can be individually controlled, this forces segments which contain arguments to contain no other information that should be protected differently, again compromising programming generality, unless segments are inexpensive enough that, as a matter of course, every data item is placed in its own segment. It may also be expensive to dynamically include and remove the argument referencing capabilities from the called ring. The third possible solution is copying arguments into segments that are accessible in the called ring, and then copying them back to their original locations on return. This solution restricts the possibility of sharing arguments with parallel processes. None of the three solutions lends itself to a straightforward hardware implementation.

The second unpleasant characteristic is that a gate must be provided for the downward return. (For an upward return the nested subset property of rings made a return gate unnecessary.) The return gate must be created at the time of the upward call and be destroyed when the subsequent return occurs. If recursive calls into a ring are allowed, then this gate must behave as though it were stored in a push-down stack, so that only the gate at the top of the stack can be used. The gates specified in SDW's seem poorly suited to this sort of dynamic behavior. Processor mechanisms to provide dynamic, stacked return gates are not obvious at this time.

Because of these two problems, the hardware described in the next section does not implement upward calls and downward returns without software intervention. Although the same object code sequences that perform all calls and returns are used in these cases as well, the hardware responds to each attempted upward call or downward return by generating a trap to a supervisor procedure which performs the necessary environment adjustments.

The manner in which the stack pointer register value of the calling procedure is saved when a call occurs and restored when the subsequent return occurs has not yet been discussed. For a same-ring or downward call, it is reasonable to trust the called procedure to save the value left in the stack pointer register by the calling procedure and then restore it before the subsequent return, since in these cases the called procedure has access capabilities which allow it to cause the calling procedure to malfunction in other ways anyway. For an upward call and the subsequent downward return, the same convention can be used without violating the protection provided by the lower ring if the intervening software verifies the restored stack pointer register value when performing the downward return.

Hardware Implementation of Rings

In this section the ideas presented in the previous sections are gathered into a description of a design for processor hardware to implement rings. The description touches upon only those aspects of the processor organization that are relevant to access control. The segmented addressing hardware described earlier serves as the foundation of the ring implementation mechanisms.

Figure 3 presents a schematic description of storage formats and processor registers that are relevant to the discussion which follows. The DBR and SDW's have already been mentioned. The three 3-bit ring numbers in an SDW (SDW.R1, SDW.R2, and SDW.R3) delimit the read, write, and execute brackets and the gate extension. The write bracket is rings 0 through SDW.R1, the execute bracket SDW.R1 through SDW.R2, and the gate extension SDW.R2+1 through SDW.R3. Rather than providing a fourth number to specify the top of the read bracket, SDW.R2 is reused for this purpose. Thus the read bracket is rings 0 through SDW.R2. Forcing the top of the read and execute brackets to coincide in this manner does not seem to preclude any important cases, and saves one ring number in the SDW. Supervisor code for constructing SDW's must guarantee that SDW.R1 $\leq$ SDW.R2 $\leq$ SDW.R3 is true. The single-bit read, write, and execute flags (SDW.R, SDW.W, and SDW.E) also appear. Finally, the list of gate locations of a segment is compressed to a single fixed-length field (SDW.GATE) by requiring all gate locations to be gathered together, beginning at location 0 of a segment. SDW.GATE contains the number of gate locations present.

The instruction pointer register (IPR) specifies the current ring of execution and the two-part address of the next instruction to be executed. The general format of an instruction word in memory (INST) is also shown for later reference.

The program accessible pointer registers (PR0, PR1, ...) each contain a two-part address and a ring number. Because segment numbers are not generally known at the time a procedure segment is compiled, machine instructions specify two-part operand addresses by giving an offset (in INST.OFFSET) relative to one of the PR's (specified by INST.PRNUM) or IPR. The ring number in a pointer register (PR*n*.RING) is used to specify a validation level for the address, and is part of the mechanism that allows an executing procedure to assume the access capabilities of a higher-numbered ring for referencing arguments. One of the PR's is intended to serve as the stack pointer register mentioned earlier.

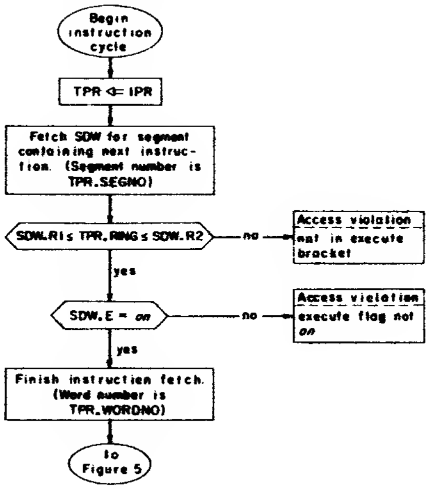
Indirect addressing may be specified in an instruction by setting the indirect flag (INST.I). Indirect words (IND) contain the same information as PR's, and may also indicate further indirection with an indirect flag (IND.I).

The final item in Figure 3 is the temporary pointer register (TPR). The TPR is an internal processor register that is not program accessible. It is used to form the two-part address of each virtual memory reference made. The ring number (TPR.RING) provides the value with respect to which permission to reference the virtual memory location is validated.

There are two aspects to the implementation of rings in hardware. The first is access checking logic, integrated with the segmented addressing hardware, that validates each virtual memory reference. The second is special instructions for changing the ring of execution. The best way to describe the first aspect is to trace the processor instruction cycle, paying particular attention to the places where operations related to access validation occur. The second aspect will be discussed when the description of the instruction cycle reaches the point where the instruction is actually performed.

The first phase of the instruction cycle, retrieving the next instruction to be executed, is described in Figure 4. At the point during address translation that the SDW for the segment containing the instruction becomes available, the ring of execution (now TPR.RING) is matched against the execute bracket defined in the SDW and the execute flag is checked. If the segment may be executed from the current ring of execution the instruction fetch is completed. The access violations and other conditions requiring software intervention shown in this and following figures generate traps, derailing the instruction cycle. A traps action is described later in this section.

Fig. 4. Retrieval of next instruction to be executed.


Begin instruction cycle
TPR ⇐ IPR
Fetch SDW for segment containing next instruction. (Segment number is TPR.SEGNO)
SDW.R1 ≤ TPR.RING ≤ SDW.R2
no
Access violation
not in execute bracket
yes
SDW.E = on
no
Access violation
execute flag not on
yes
Finish instruction fetch. (Word number is TPR.WORDNO)
to Figure 5

The next phase of the instruction cycle, calculating in TPR the effective address of the instruction's operand, is described in Figure 5. This phase occurs only if the instruction has an operand in memory. The effective address is the final two-part address of the operand (after all address modifications and indirections have taken place) together with an effective ring number which is used to validate the actual reference to the operand.

The formation of a two-part address in TPR.SEGNO and TPR.WORDNO is very straightforward and is described by Figure 5. The calculation of the ring number portion of the effective address in TPR.RING and the access validation performed before retrieving indirect words, also shown in Figure 5, need further comment.

The effective ring portion of the effective address provides a procedure with the means of voluntarily assuming the access capabilities of a higher-numbered ring when making an instruction operand reference. The effective ring number also records the highest-numbered ring from which a procedure (in the same process) possibly could have influenced the effective address calculation. The first opportunity for the value of TPR.RING to change during effective address calculation occurs if the instruction contains an address that is an offset relative to some PR*n*. In this case TPR.RING is updated with the larger of its current values (still the current ring of execution) and the ring number in the specified pointer register (PR*n*.RING). Thus, if PR*n*.RING contains a value that is greater than the current ring of execution, validation of the operand reference will be as though execution were occurring in this higher-numbered ring.

Fig. 5. Formation in TPR of effective address of instruction operand.

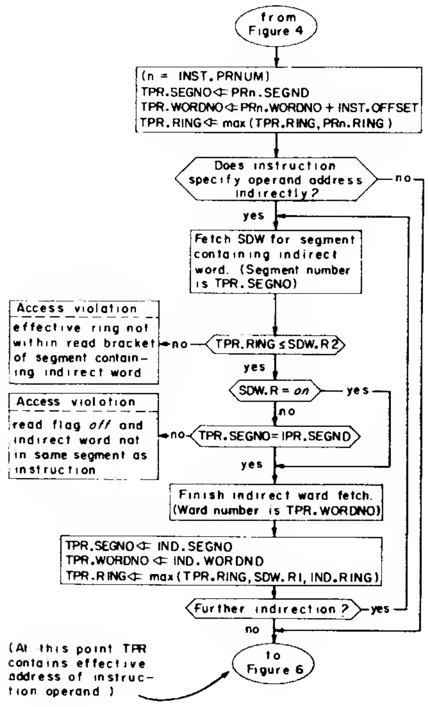
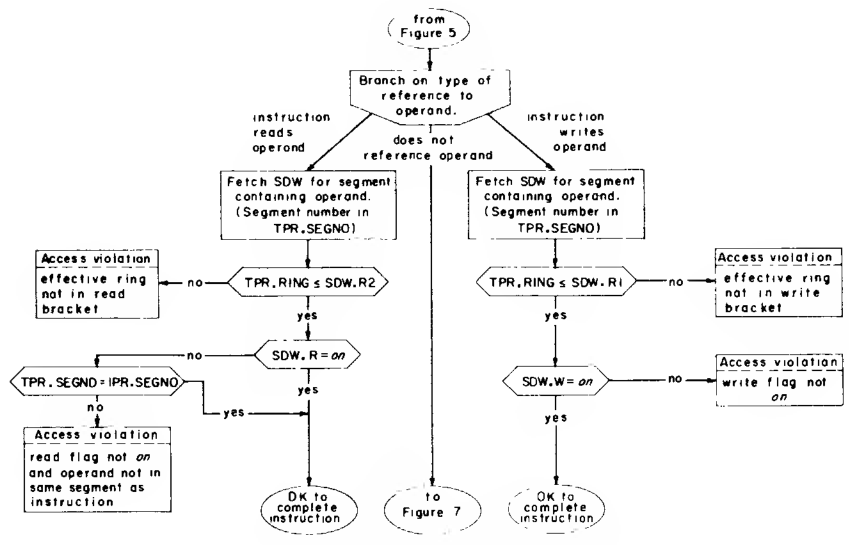

from Figure 4
(n = INST.PRNUM)
TPR.SEGNO ⇐ PRn.SEGNO
TPR.WORDNO ⇐ PRn.WORDNO + INST.OFFSET
TPR.RING ⇐ max (TPR.RING, PRn.RING)
Does instruction specify operand address indirectly?
no
yes
Fetch SDW for segment containing indirect word. (Segment number is TPR.SEGNO)
Access violation
effective ring not within read bracket of segment containing indirect word
no
TPR.RING ≤ SDW.R2
yes
SDW.R = on
yes
no
Access violation
read flag off and indirect word not in same segment as instruction
no
TPR.SEGNO = IPR.SEGNO
yes
Finish indirect word fetch. (Word number is TPR.WORDNO)
TPR.SEGNO ⇐ IND.SEGNO
TPR.WORDNO ⇐ IND.WORDNO
TPR.RING ⇐ max (TPR.RING, SDW.R1, IND.RING)
Further indirection ?
yes
no
(At this point TPR contains effective address of instruction operand)
to Figure 6

Fig. 6. Access validation for instructions which read or write their operands.


from Figure 5
Branch on type of reference to operand.
instruction reads operand
does not reference operand
instruction writes operand
Fetch SDW for segment containing operand. (Segment number in TPR.SEGNO)
Fetch SDW for segment containing operand. (Segment number in TPR.SEGNO)
Access violation
effective ring not in read bracket
no
TPR.RING ≤ SDW.R2
yes
TPR.RING ≤ SDW.R1
no
Access violation
effective ring not in write bracket
yes
no
SDW.R = on
yes
TPR.SEGNO = IPR.SEGNO
no
yes
Access violation
read flag not on and operand not in same segment as instruction
SDW.W = on
no
Access violation
write flag not on
yes
OK to complete instruction
to Figure 7
OK to complete instruction

The remaining opportunities to change the value of TPR.RING occur in conjunction with the processing of indirect words involved in the effective address calculation. Each time an indirect word is retrieved, TPR.RING is updated with the larger of its current values, the ring number in the indirect word (IND.RING), and the top of the write bracket for the segment containing the indirect word (SDW.R1). The ring number in the indirect word has the same purpose as the ring number in a pointer register—forcing validation of the operand reference relative to some higher-numbered ring. Including in the calculation the top of the write bracket of the segment containing the indirect word, however, has another purpose. The top of the write bracket represents the highest-numbered ring from which a procedure in the same process could have altered the indirect word and thereby influenced the result of the effective address calculation. Taking into account SDW.R1 when updating TPR.RING guarantees that the operand reference will be validated with respect to the highest-numbered ring which could have influenced the effective address.

The capability to read an indirect word during effective address formation must be validated before the indirect word is retrieved. Validation is with respect to the value in TPR.RING at the time the indirect word is encountered. At the conclusion of the effective address calculation described in Figure 5, TPR contains the effective address of the instruction operand, including the effective ring number with respect to which the reference to the operand will be validated.

The next phase of the instruction cycle is to perform the instruction. For the purpose of access validation, the possible instructions may be broken into three groups, according to the type of reference made to the operand. Figure 6 shows the access validation for the straightforward cases of instructions which read their operands and instructions which write their operands. The third group, instructions which do not reference their operands, is illustrated in Figure 7. One set in this group is the "Effective Address to Pointer Register"-type (EAP-type) instructions which load the RING, SEGNO, and WORDNO fields of PR*n* with the corresponding fields of TPR. The operand is not referenced, so no access validation is required. Instructions of this type are important, as will be seen later, for they are the only way to load PR's.

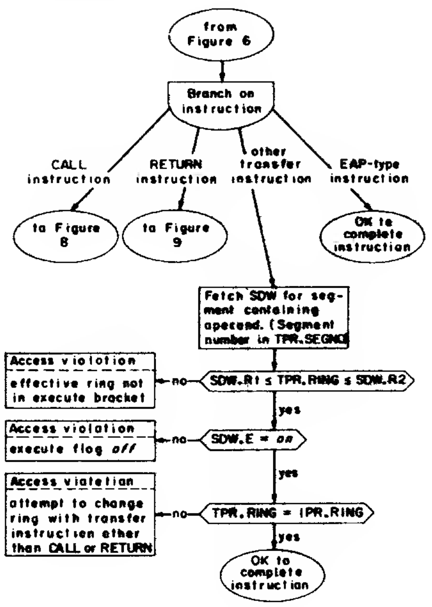
The remaining instructions illustrated in Figure 7 are transfer instructions. To provide some protection against changing the ring of execution by accident, all transfer instructions except two, CALL and RETURN, are constrained from doing so. Since a transfer instruction does not reference its operand, but just loads the address of its operand into the instruction counter, no access validation is really required. However, an advance check on whether reloading IPR from TPR will result in an access violation when the next instruction is retrieved is very useful from the standpoint of debugging, for it catches the access violation while it is still possible to identify the instruction which made the illegal transfer. Figure 7 describes the advance check for transfer instructions other than CALL and RETURN.

The two instructions that remain to be considered are the instructions which can change the ring of execution: CALL and RETURN. They are intended to be used to implement the same-named linguistic operations.[1] CALL will automatically switch the ring of execution to a lower number and RETURN to a higher number if the occasion requires it. These instructions also function properly for calls and returns within the same ring. When used to perform an upward call or a downward return, the instructions cause traps which allow software intervention.

Figure 8 describes the access validation and performance of the CALL instruction. Several points require further explanation. The first concerns gates. From Figure 8 it is apparent that a CALL must be directed at a gate location even when the called procedure will execute in the same ring as the calling procedure. The rationale for this use of the gate list of a segment is that it can provide protection against accidental calls to locations that are not entry points, even when the call comes from within the same ring. Thus, SDW.GATE for a procedure segment usually specifies the number of externally defined entry points in the procedure segment. These become gates for higher-numbered rings in the sense described in the previous sections only if the top of the gate extension of the segment is above the top of the execute bracket, i.e. only if SDW.R3 > SDW.R2 for the segment. The price paid for this error detection ability is that if any externally defined entry point in a procedure segment is a gate for a higher-numbered ring,

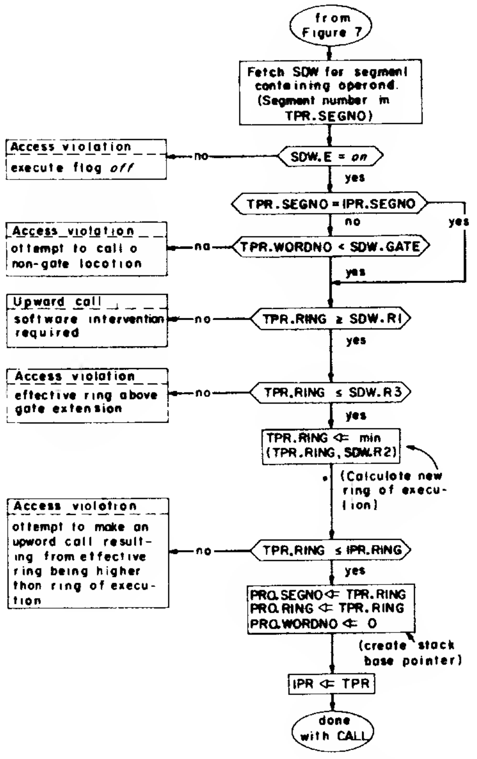
[1] RETURN may also be used to implement the nonlocal goto operation.

Fig. 7. Access validation for instructions which do not reference their operands.



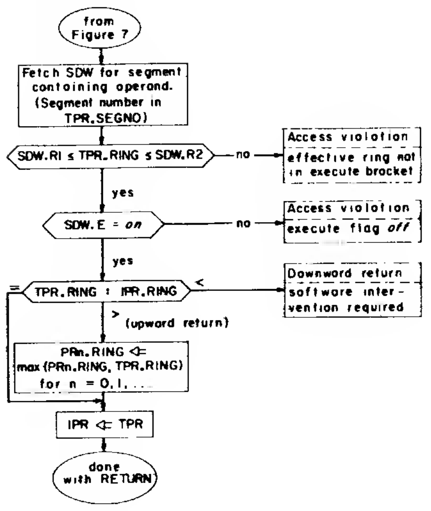
from Figure 6
Branch on instruction
CALL instruction
RETURN instruction
other transfer instruction
EAP-type instruction
to Figure 8
to Figure 9
OK to complete instruction
Fetch SDW for segment containing operand. (Segment number in TPR.SEGNO)
Access violation
effective ring not in execute bracket
no
SDW.R1 ≤ TPR.RING ≤ SDW.R2
yes
Access violation
execute flag off
no
SDW.E = on
yes
Access violation
attempt to change ring with transfer instruction other than CALL or RETURN
no
TPR.RING = IPR.RING
yes
OK to complete instruction

Fig. 8. Access validation and performance of the CALL instruction.



from Figure 7
Fetch SDW for segment containing operand. (Segment number in TPR.SEGNO)
Access violation
execute flag off
no
SDW.E = on
yes
TPR.SEGNO = IPR.SEGNO
no
yes
Access violation
attempt to call a non-gate location
no
TPR.WORDNO < SDW.GATE
yes
Upward call
software intervention required
no
TPR.RING ≥ SDW.R1
yes
Access violation
effective ring above gate extension
no
TPR.RING ≤ SDW.R3
yes
TPR.RING ⇐ min (TPR.RING, SDW.R2)
(Calculate new ring of execution)
Access violation
attempt to make an upward call resulting from effective ring being higher than ring of execution
no
TPR.RING ≤ IPR.RING
yes
PRO.SEGNO ⇐ TPR.RING
PRO.RING ⇐ TPR.RING
PRO.WORDNO ⇐ 0
(create stack base pointer)
IPR ⇐ TPR
done with CALL

Fig. 9. Access validation and performance of the RETURN instruction.


from Figure 7
Fetch SDW for segment containing operand. (Segment number in TPR.SEGNO)
SDW.R1 ≤ TPR.RING ≤ SDW.R2
no
Access violation
effective ring not in execute bracket
yes
SDW.E = on
no
Access violation
execute flag off
yes
TPR.RING : IPR.RING
=
<
Downward return
software intervention required
> (upward return)
PRn.RING ⇐ max (PRn.RING, TPR.RING) for n = 0, 1, . . .
IPR ⇐ TPR
done with RETURN

then all are. On intersegment transfers of control within the same ring, the gate restriction can be bypassed by using a normal transfer instruction rather than a CALL. The only exception to having the CALL instruction respect the gate list of the operand segment occurs if the operand is in the same segment as the instruction. Allowing a CALL instruction to ignore the gate list of the segment containing the instruction permits it to be used to implement calls to internal procedures.

The access validation for the CALL instruction is made relative to the ring number computed as part of the effective address. Since, as a result of PR-relative addressing and indirection, the effective ring value (TPR.RING) can be higher than the current ring of execution (IPR.RING), what would appear to be a call within the same ring or to a lower ring with respect to TPR.RING can in fact be an upward call with respect to IPR.RING. Because in normal circumstances this situation represents an error, the decision is made to generate an access violation when it occurs, even if the current ring of execution is within the execute bracket of the called procedure segment.

CALL generates in PR0 a pointer to word 0 of the stack segment for the new ring of execution. (The PR to use as this stack base pointer is chosen arbitrarily.) The stack segment selection rule illustrated in Figure 8 is that the segment number of the appropriate stack segment is the same as the new ring number.[2] The final transfer of control is achieved by reloading IPR.RING, IPR.SEGNO, and IPR.WORDNO from the corresponding fields of TPR.

The RETURN instruction is described by Figure 9. The access validation is the same as for other transfer instructions. The ring to which the return is made is specified by the effective ring portion of the effective address generated by the RETURN instruction. In the case that the return is upward, the ring number fields in all pointer registers are replaced with the larger of their current values and the new ring of execution. This replacement, together with the fact that PR's can only be loaded with EAP-type instructions, guarantees that PR*n*.RING can never contain a value that is less than IPR.RING, a fact which proves very useful when passing arguments on a downward call and which makes it easy to perform an upward return to the proper ring. (See the next section for details.)

Two items remain to be considered to complete the description of the processor hardware for implementing rings. One is the action of a trap. Traps are generated by a variety of conditions in Figures 4–9, as well as by missing segments and pages, I/O completions, etc. When the processor detects such a condition, it changes the ring of execution to zero and transfers control to a fixed location in the supervisor. A special instruction allows the state of the processor at the time of the trap to be restored later if appropriate, resuming the disrupted instruction.

The other item concerns privileged instructions. Certain instructions, if executable by all procedure segments, could invalidate the protection provided by the ring mechanisms. Among these are the instructions to load the DBR, start I/O, and restore the processor state after a trap. Such instructions are designated as privileged and will be executed by the processor only in ring 0. This convention restricts their use to supervisor procedures.

Call and Return Revisited

The intended use of the hardware mechanisms just described is illustrated by considering again two key aspects of the linguistic meaning of the operations call and return.

[2] Two subtle features may be included at this point by using a more sophisticated stack segment selection rule. If the CALL instruction does not change the ring of execution, then the segment number for the stack base pointer is taken directly from the stack pointer register, allowing the continued use of a nonstandard stack segment for procedures executing in the same ring. If the CALL instruction does change the ring of execution then the new stack segment number is calculated by adding the new ring number to an additional DBR field that specifies the eight consecutively numbered segments that are the standard stack segments of the process. The use of the additional DBR field allows more flexibility in stack segment assignment, facilitating the preservation of stack history following an error and the implementation of forked stacks.

The first aspect to be reconsidered is the way arguments are passed and referenced. A procedure making a call constructs an array of indirect words containing the addresses of the various arguments to be passed with the call. To inform the called procedure of the location of this argument list, the calling procedure loads a specific PR designated by software convention (call it PR*a*) with the address of the beginning of the argument list. An instruction of the called procedure can reference the *n*th argument as its operand by using an indirect address. The location of the indirect word is specified in the instruction as PR*a* offset by *n*. If this operand reference constitutes an upward cross-ring argument reference then the proper validation is automatic, for PR*a*.-RING, as set by the calling procedure, must contain a number that is greater than or equal to the number of the ring in which the calling procedure was executing when the call was made. Thus, validation of all argument references by the called procedure will be with respect to an effective ring that is at least as high as the ring of the caller.

The ring number in PR*a*, then, allows the called procedure to automatically assume the fewer access capabilities of the calling procedure in the case of an upward cross-ring argument reference via PR*a* and the argument list. Not all argument references, however, will be made in this way. For example, if an argument is an array, then the corresponding argument list indirect word will address the first element. The called procedure may find it convenient to load some free PR, say PR1, with the actual two-part address of the beginning of that array argument so that array indexing can be more easily accomplished. If PR1 is loaded with an EAP-type instruction whose operand address is specified via PR*a* and the argument list, then the proper effective ring number will automatically be put in PR1.RING, and subsequent references to the argument via PR1 will also be validated with respect to an effective ring that is at least as high as the ring of the caller. If PR1 is then stored as an indirect word, this effective ring is put into the RING field of the indirect word. In fact, as long as the called procedure does not make an explicit effort to lower the effective ring associated with an argument address, e.g. by zeroing the RING field of an indirect word, then all manipulations of the argument address are safe, and all argument references will be validated with respect to an effective ring that is at least as high as the ring of the caller.[3]

The second aspect to be reconsidered with respect to call and return is the way in which a return to the proper ring is accomplished. As described earlier, the hardware guarantees that the RING fields in all PR's always contain values greater than or equal to the current ring of execution. Thus, after a call all PR's except PR0, which is altered by the CALL instruction, initially contain the ring of the caller (or some higher number) in their RING fields. It follows that any scheme for returning which depends upon one of these values is secure. For example, the convention described earlier for restoring the stack pointer register value of the caller before a return makes it natural to address the operand of the RETURN instruction via this restored PR. (For this scheme to work, the return point must have been saved by the caller at a standard position in its stack area before the call occurred.) The RETURN instruction is thus guaranteed to generate an effective ring number no lower than the ring of the calling procedure and therefore will return control to the ring of the caller or some higher-numbered ring.

Use of Rings

Some insight into the functional capabilities of rings can be gained by considering briefly the way the basic mechanisms described in the previous sections are used in Multics.

The ring protection scheme allows a layered supervisor. to be included in the virtual memory of each process. In Multics, the lowest-level supervisor procedures, such as those implementing the primitive operations of access control, I/O, memory multiplexing, and processor multiplexing, execute in ring 0. The remaining supervisor procedures execute in ring 1. Examples of ring 1 supervisor procedures are those performing accounting, input/output stream management, and file system search direction. (Deciding how many layers to use and which procedures should execute in each layer is an interesting engineering design problem.) Supervisor data segments have read and write brackets that end at ring 0 or ring 1, depending on which layer of the supervisor needs to access each.

Implicit invocation of certain ring 0 supervisor procedures occurs as a result of a trap. Explicit invocation of selected ring 0 and ring 1 supervisor procedures by procedures executing in rings 2–5 of a process is by standard subroutine calls to gates. Procedures executing in rings 6 and 7 are not given access to supervisor gates.

Because separate access control lists for each segment and separate descriptor segments for each process provide the means to control separately the use of each segment by each user's process, not all gates into supervisor rings need be available to the processes of all users, and not all gates need have the same gate extension associated with them. For example, some gates into ring 0 are accessible to the processes of all users, but only to procedures executing in ring 1. Such gates provide the internal interfaces between the two layers of the super-

[3] This property allows the correct argument validation to occur naturally when an argument is passed along a chain of downward calls. The RING field of an argument list indirect word will specify the ring which originally provided the argument. If this value is higher than the value of PR*a*.RING, then the indirect word ring number will become the effective ring for validation of references to the corresponding argument.

visor. Some gates into ring 1 are accessible to procedures executing in rings 2–5 in the processes of selected users, but are not accessible at all from the processes of other users. An example of the latter kind is a gate for registering new users that is available only from the processes of system administrators.

As pointed out by Dijkstra [6], a layered supervisor has several advantages. Constructing the supervisor in layers enforced by ring protection reinforces these advantages. It limits the propagation of errors, thereby making the supervisor easier to modify correctly and increasing the level of confidence that the supervisor functions correctly. For example, changes can be made in ring 1 without having to recertify the correct operation of the procedures in ring 0.

By arranging for standard user procedures to execute in ring 4, rings 2 and 3 become available for the protection of user-constructed subsystems. Subsystems executing in rings 2 and 3 of a process can be protected from procedures executing in rings 4–7 in the same way that the supervisor is protected from procedures executing in rings 2–7. All comments made about a supervisor implemented in rings 0 and 1 of each process apply to protected subsystems implemented in rings 2 and 3. Different protected subsystems may be operated simultaneously in rings 2 and 3 of different processes and several processes may share the use of the same protected subsystem simultaneously. The ring protection scheme allows the operation of user-constructed protected subsystems without auditing them for inclusion in the supervisor. (The software facility that forces standard user procedures to execute in ring 4, and yet allows all users to freely provide ring 3 protected subsystems for one another, is not discussed here.) Examples of protected subsystems that might be provided by various users are a proprietary compiler or a subsystem to provide interpretive access to some sensitive data base and safely log each request for information.

With most user procedures executing in ring 4, rings 5, 6, and 7 are available for user self-protection. For example, a user may debug a program by executing it in ring 5, where only procedure and data segments intended to be referenced by the program would be made accessible. The ring protection mechanisms would detect many of the addressing errors that could be made by the program and would prevent the untested program from accidently damaging other segments accessible from ring 4. In the same way ring 5 can be used for the execution of an untrusted program borrowed from another user.

Because supervisor gates are not accessible from rings 6 and 7 of any process in Multics, procedures executed in these rings have no explicit access to supervisor functions; they may, however, be given permission to call user-provided gates into rings 4 or 5. Ring 6 of a process might be used, for example, to provide a suitably isolated environment for student programs being evaluted by a grading program executing in ring 4.

The complete description of a software access control facility based on rings that allows them to be used in the manner just outlined would require another paper. A fundamental constraint enforced by this software facility is that a program executing in ring n cannot specify R1, R2, or R3 values of less than n in an access control list entry of any segment. Although a given ring may simultaneously protect different subsystems in different processes, each ring of each process can protect only one subsystem at a time. A usable software access control facility must constrain each user's ability to dynamically set and modify access control specifications so that this sole occupant property can be verified and enforced when necessary.

Conclusions

The hardware mechanisms derived and described in this paper implement a methodical generalization of the traditional supervisor/user protection scheme that is compatible with a shared virtual memory based on segmentation. This generalization solves three significant kinds of problems of a general purpose system to be used as a computer utility:

- users can create arbitrary, but protected, subsystems for use by others;
- the supervisor can be implemented in layers which are enforced;
- the user can protect himself while debugging his own (or borrowed) programs.

The subset access property of rings of protection does not provide for what may be called "mutually suspicious programs" operating under the control of a single process. On the other hand, it is just that subset property which imposes an organization which is easy to understand and thus allows a system or subsystem designer to convince himself that his implementation is complete. Also, it is just the subset property which is the basis for a hardware implementation that is integrated with segmentation mechanisms, requiring very small additional costs in hardware logic and processor speed.

The long-range effect of hardware protection mechanisms which permit calls to protected subsystems that use the same mechanisms as calls to other procedures is bound to be significant. In the interface to the supervisor of most systems there are many examples of facilities whose interface design is biased by the assumption that a call to the supervisor is relatively expensive; the usual result is to place several closely related functions together in the supervisor, even though only one of the group really needs protection. For example, in the Multics typewriter I/O package, only the functions of copying data in and out of shared buffer areas and of executing the privileged instruction to initiate I/O channel operation need to be protected. But, since these two functions are deeply tangled with typewriter operation strategy and code conversion, the typewriter I/O control package is currently implemented as a set of procedures all located in the lowest-numbered ring of the system, thus increasing the quantity of code which has maximum privilege.

A similar example is found in many file system designs, where complex file search operations are carried out entirely by protected supervisor routines rather than by unprotected library packages, primarily because a complex file search requires many individual file access operations, each of which would require transfer to a protected service routine, which transfer is presumed costly.

The initial version of Multics used software implemented rings of protection. The result was a very conservative use of the rings: originally just two supervisor rings and one user ring were employed, and the two supervisor rings were temporarily collapsed into one (thus exploiting the programming generality objective referred to before) while the software ring crossing mechanisms were tuned up. Today, although there are many obvious applications waiting, the ability to use more than two rings in a computation is just beginning to be exploited. The availability with the new Multics processor of hardware implemented rings which make downward calls and upward returns no more complex than calls and returns in the same ring should significantly increase such exploitation.

Acknowledgments. The concepts embodied in the mechanisms described here were the result of seven years of maturing of ideas suggested by many workers. The original idea of generalizing the supervisor/user relationship to a multiple ring structure was suggested by R.M. Graham, E.L. Glaser and F.J. Corbató. An initial software implementation of rings using multiple descriptor segments [14] was worked out by Graham and R.C. Daley, and constructed by members of the Multics system programming team. That implementation makes use of hardware access mode indicators stored in the segment descriptor word of the Honeywell 645 computer. Graham [9], in 1967, proposed a partial hardware implementation of rings of protection which included three ring numbers embedded in segment descriptor words, and a processor ring register, but which still required software intervention on all ring crossings. Though a related scheme was implemented in the Hitac 5020 time-sharing system [15], this hardware scheme was never implemented in Multics, which today (1971) still uses a version of the software implementation of rings. The complete automation of downward calls and upward returns was proposed in a thesis in 1969 [16]; the description in this paper extends that thesis slightly with the addition of ring numbers to indirect words and the processor pointer registers, as suggested by Daley. The CALL and RETURN instructions proposed there have also been simplified.

The hardware implemented call and return, and automatically managed stacks, were at least partly inspired by similar mechanisms which have long been used on computer systems of the Burroughs Corporation [4, 11].

In addition to those named above, D.D. Clark, C.T. Clingen, R.J. Feiertag, J.M. Grochow, N.I. Morris, M.A. Padlipsky, M.R. Thompson, V.L. Voydock, and V.A. Vyssotsky contributed significant help in understanding and implementing rings of protection.

References

1. Apfelbaum, H., and Oppenheimer, G. Design of virtual memory systems. Proc. 1971 IEEE Internat. Comput. Soc. Conf., Boston, pp. 115–116.
2. Arden, B.W., et al. Program and addressing structure in a time-sharing environment. *J. ACM 13*, 1 (Jan. 1966), 1–16.
3. Bensoussan, A., Clingen, C.T., and Daley, R.C. The Multics virtual memory. Proc. Second ACM Symposium on Operating Systems Principles. Princeton, N.J., 1969, ACM New York, 1971, pp. 30–42 Also *Comm. ACM* (to appear).
4. Burroughs Corporation. A Narrative Description of the Burroughs B5500 Master Control Program. Detroit, Mich. Oct. 1969.
5. Dennis, J.B., and VanHorn, E.C. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (Mar. 1966), 143–155.
6. Dijkstra, E.W. The structure of the "THE"- multiprogramming system. *Comm. ACM 11*, 5 (May 1968), 341–346.
7. Evans, D.C., and LeClerc, J.Y. Address mapping and the control of access in an interactive computer. Proc. AFIPS 1967 SJCC, Vol. 30, AFIPS Press, Montvale, N.J. pp. 23–30.
8. Fabry, R.S. Preliminary description of a supervisor for a computer organized around capabilities. Quarterly Progress Rep. No. 18, Institute of Computer Research, U. of Chicago, I–B 1–97.
9. Graham, R.M. Protection in an information processing utility. *Comm. ACM 11*, 5 (May 1968), 365–369.
10. Honeywell Information Systems Inc., Model 645 Processor Reference Manual. Cambridge Information Systems Laboratory, Apr. 1971.
11. Hauck, E.A., and Dent, B.A. Burrough's B6500/B7500 stack mechanisms. Proc. AFIPS 1968 SJCC, Vol. 32, AFIPS Press, Montvale, N.J. pp. 245–251.
12. Lampson, B.W. An Overview of the CAL Time-Sharing System. Computation Center, U. of California, Berkeley, Sept. 1969.
13. Lampson, B.W. Dynamic protection structures. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., 27–38.
14. MIT Project MAC. Multics Programmer's Manual. 1969.
15. Motobayashi, S., Masuda, T., and Takahashi, N. The Hitac 5020 time-sharing system. Proc ACM 24th Nat. Conf. 1969, ACM New York, pp. 419–429.
16. Schroeder, M.D. Classroom model of an information and computing service. S.M. Th. MIT, Dep. Elec. Eng., Feb. 1969. [Expanded version available as Proj. MAC Tech. Rep. MAC–TR–80.]
17. Vanderbilt, D.H. Controlled information sharing in a computer utility. MIT Project MAC, MAC–TR–67, 1969.

The Multics PL/I Compiler

by R.A. Freiburghouse. Reprinted from AFIPS Conference Proceedings 35, AFIPS Press, 1969, pp. 187-199, with permission. Copyright 1969 by AFIPS Press.

This paper describes the second PL/I compiler successfully constructed for Multics, and used for the compilation of the operating system itself. Although today a third and better PL/I compiler is now in use, the basic organization of the second compiler was preserved. Probably the most significant observation about these two compilers is that even though they implement the full language, they generate object code of high enough quality (often better than an average machine language programmer) to be used in the operating system itself. Since the concept of writing the system in PL/I, to make its description smaller, more maintainable, and easier to learn, was considered pivotal in the goals of Multics, this paper is especially significant.

The multics PL/1 compiler

by R. A. FREIBURGHOUSE

General Electric Company
Cambridge, Massachusetts

INTRODUCTION

The Multics PL/1 compiler is in many respects a "second generation" PL/1 compiler. It was built at a time when the language was considerably more stable and well defined than it had been when the first compilers were built.[1,2] It has benefited from the experience of the first compilers and avoids some of the difficulties which they encountered. The Multics compiler is the only PL/1 compiler written in PL/1 and is believed to be the first PL/1 compiler to produce high speed object code.

The language

The Multics PL/1 language is the language defined by the IBM "PL/1 Language Specifications" dated March, 1968.[1] At the time this paper was written most language features were implemented by the compiler but the run time library did not include support for input and output, as well as several lesser features. Since the multi-tasking primitives provided by the Multics operating system were not well suited to PL/1 tasking, PL/1 tasking was not implemented. Inter-process communication (Multics tasking) may be performed through calls to operating system facilities.

The system environment

The compiler and its object programs operate within the Multics operating system.[3,4,5] The environment provided by this system includes a virtual two dimensional address space consisting of a large number of segments. Each segment is a linear address space whose addresses range from 0 to 64K. The entire virtual store is supported by a paging mechanism which is invisible to the program. Each program operating in this environment consists of two segments: a text segment containing a pure re-entrant procedure, and a linkage segment containing out-references (links), definitions (entry names), and static storage local to the program. The text segment of each program is sharable by all other users on the system. Linking to a called program is normally done dynamically during program execution.

Implementation techniques

The entire compiler and the Multics operating system were written in EPL, a large subset of PL/1 containing most of the complex features of the language. The EPL compiler was built by a team headed by M. D. McIlroy and R. Morris of Bell Telephone Laboratories. Several members of the Multics PL/1 project modified the original EPL compiler to improve its object code performance, and utilized the knowledge acquired from this experience in the design of the Multics PL/1 compiler. EPL and Multics PL/1 are sufficiently compatible to allow the Multics PL/1 compiler to compile itself and the operating system.

The Multics PL/1 compiler was built and de-bugged by four experienced system programmers in 18 months. All program preparation was done on-line using the CTSS time-sharing system at MIT. Most de-bugging was done in a batch mode on the GE645, but final de-bugging was done on-line using Multics.

The extremely short development time of 18 months was made possible by these powerful tools. The same design programmed in a macro-assembly language using card input and batched runs would have required twice as much time, and the result would have been extremely unmanageable.

Design objectives

The project's design decisions and choice of techniques were influenced by the following objectives:

1. A correct implementation of a reasonably complete PL/1 language.
2. A compiler which produced relatively fast object code for all language constructs. For similar language constructs, the object code was expected to equal or exceed that produced by most Fortran or COBOL compilers.
3. Object program compatibility with EPL object programs and other Multics languages.
4. An extensive compile time diagnostic facility.
5. A machine independent compiler capable of bootstrapping itself onto other hardware.

The compiler's size and speed were considered less important than the above mentioned objectives. Each phase of the original compiler occupies approximately 32K, but after the compiler has compiled itself that figure will be about 24K. The original compiler was about twice as slow as the Multics Fortran compiler. The bootstrapped version of the PL/1 compiler is expected to be considerably faster than the original version but it will probably not equal the speed of Fortran.

An overview of the compiler

The Multics PL/1 compiler is designed along traditional lines. It is not an interactive compiler nor does it perform partial compilations. The compiler translates PL/1 external procedures into relocatable binary machine code which may be executed directly or which may be bound together with other procedures compiled by any Multics language processor.

The notion of a phase is particularly useful when discussing the organization of the Multics PL/1 compiler. A phase is a set of procedures which performs a major logical function of compilation, such as syntactic analysis. A phase is not necessarily a memory load or a pass over some data base although it may, in some cases, be either or both of these things.

The dynamic linking and paging facilities of the Multics environment have the effect of making available in virtual storage only those specific pages of those particular procedures which are referenced during an execution of the compiler. A phase of the Multics PL/1 compiler is therefore only a logical grouping of procedures which may call each other. The PL/1 compiler is organized into five phases: Syntactic Translation, Declaration Processing, Semantic Translation, Optimization, and Code Generation.

The internal representation

The internal representation of the program being compiled serves as the interface between phases of the compiler. The internal representation is organized into a modified tree structure (the program tree) consisting of nodes which represent the component parts of the program, such as blocks, groups, statements, operators, operands, and declarations. Each node may be logically connected to any number of other nodes by the use of pointers.

Each source program block is represented in the program tree by a block node which has two lists connected to it: a statement list and a declaration list. The elements of the declaration list are symbol table nodes representing declarations of identifiers within that block. The elements of the statement list are nodes representing the source statements of that block. Each statement node contains the root of a computation tree which represents the operations to be performed by that statement. This computation tree consists of operator nodes and operand nodes.

The operators of the internal representation are n-operand operators whose meaning closely parallels that of the PL/1 source operators. The form of an operand is changed by certain phases, but operands generally refer to a declaration of some variable or constant. Each operand also serves as the root of a computation tree which describes the computations necessary to locate the item at run time.

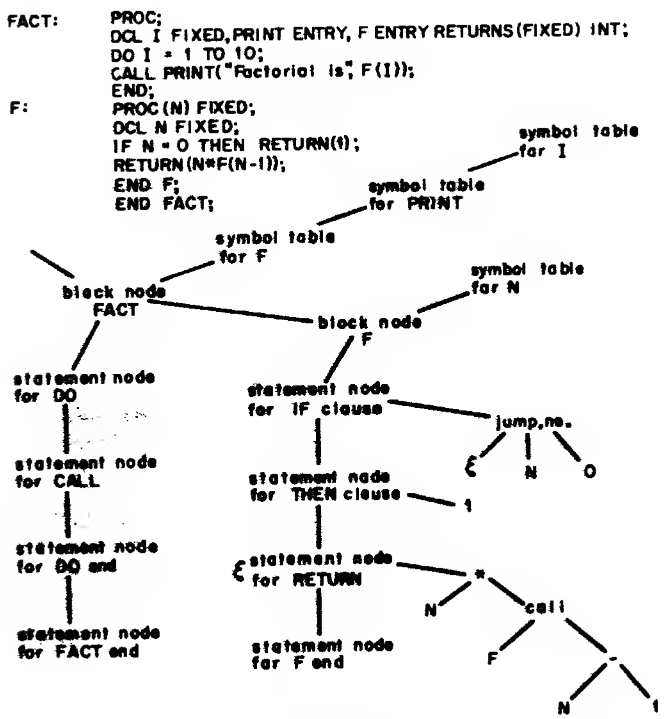
This internal representation is machine independent in that it does not reflect the instruction set, the addressing properties, or the register arrangement of the GE645. The first four phases of the compiler are also machine independent since they deal only with this machine independent internal representation. Figure 1 shows the internal representation of a simple program.

Syntactic translation

Syntactic analysis of PL/1 programs is slightly more difficult than syntactic analysis of other languages such as Fortran. PL/1 is a larger language containing more syntactic constructs, but it does not present any significantly new problems. The syntactic translator consists of two modules called the lexical analyzer and the parse.

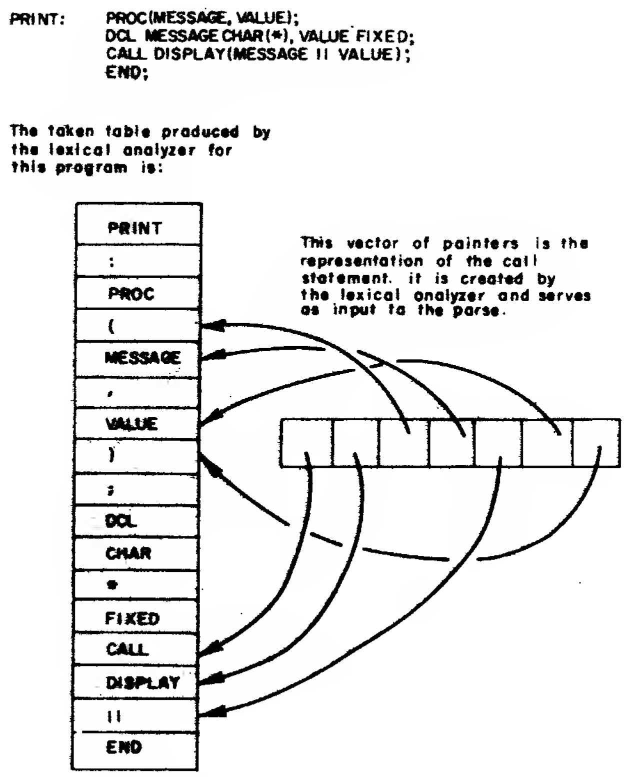
Lexical analysis

The lexical analyzer organizes the input text into groups of tokens which represent a statement. It also creates the source listing file and builds a token table which contains the source representation of all tokens in



```
FACT:    PROC;
         DCL I FIXED,PRINT ENTRY, F ENTRY RETURNS(FIXED) INT;
         DO I = 1 TO 10;
         CALL PRINT("Factorial is", F(I));
         END;
F:       PROC(N) FIXED;
         DCL N FIXED;
         IF N = 0 THEN RETURN(1);
         RETURN(N*F(N-1));
         END F;
         END FACT;
```

Figure 1—The internal representation of a program. The example is greatly simplified. Only the statements of procedure *F* are shown in detail.



```
PRINT:   PROC(MESSAGE, VALUE);
         DCL MESSAGE CHAR(*), VALUE FIXED;
         CALL DISPLAY(MESSAGE || VALUE);
         END;
```

The token table produced by the lexical analyzer for this program is:
PRINT
:
PROC
(
MESSAGE
,
VALUE
)
;
DCL
CHAR
*
FIXED
CALL
DISPLAY
||
END
This vector of pointers is the representation of the call statement. it is created by the lexical analyzer and serves as input to the parse.

Figure 2—The output of the lexical analyzer.

the source program. A token is an identifier, a constant, an operator or a delimiter. The lexical analyzer is called by the parse each time the parse wants a new statement.

The lexical analyzer is an approximation to a finite state machine. Since the lexical analyzer must produce output as well as recognize tokens, action codes are attached to the state transitions of the finite state machine. These action codes result in the concatenation of individual characters from the output until a recognized token is formed. Constants are not converted to their internal format by the lexical analyzer. They are converted by the semantic translator to a format which depends on the context in which the constant appears.

The token table produced by the lexical analyzer contains a single entry for each unique token in the source program. Searching of the token table is done utilizing a hash coded scheme which provides quick access to the table. Each token table entry contains a pointer which may eventually point to a declaration of the token. For each statement, the lexical analyzer builds a vector of pointers to the tokens which were found in the statement. This vector serves as the input to the parse. Figure 2 shows a simple example of lexical analysis.

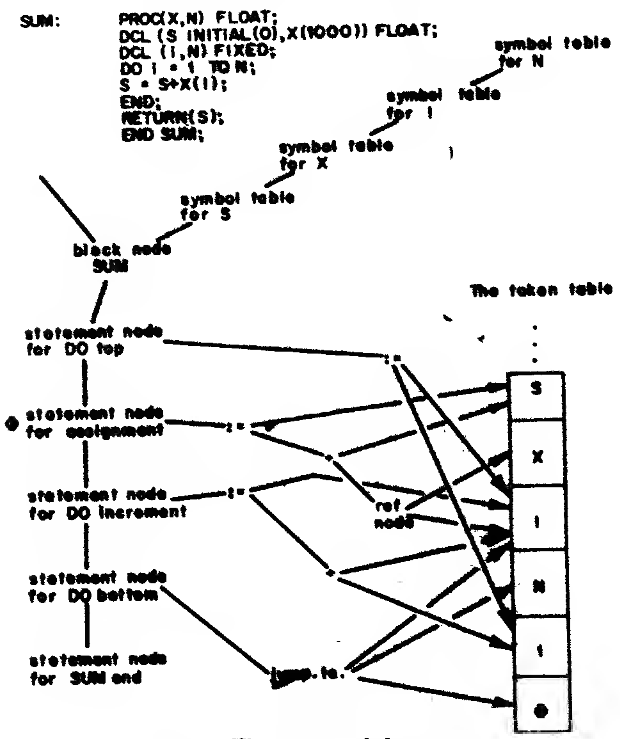
The parse

The parse consists of a set of possibly recursive procedures, each of which corresponds to a syntactic unit of the language. These procedures are organized to perform a top down analysis of the source program. As each component of the program is recognized, it is transformed into an appropriate internal representation. The completed internal representation is a program tree which reflects the relationships between all of the components of the original source program. Figure 3 shows the results of the parse of a simple program.

Syntactic contexts which yield declarative information are recognized by the parse, and this information is passed to a module called the context recorder which constructs a data base containing this information. Declare statements are parsed into partial symbol table nodes which represent declarations.

The problem of backup

The top down method of syntactic analysis is used because of its simplicity and flexibility. The use of a simple statement recognition algorithm made it possible



```
SUM:    PROC(X,N) FLOAT;
        DCL (S INITIAL(0),X(1000)) FLOAT;
        DCL (I,N) FIXED;
        DO I = 1 TO N;
        S = S+X(I);
        END;
        RETURN(S);
        END SUM;
```

symbol table for N
symbol table for I
symbol table for X
symbol table for S
block node SUM
The token table
statement node for DO top
statement node for assignment
statement node for DO increment
statement node for DO bottom
statement node for SUM end
:=
:=
:=
+
+
ref node
jump. to.
S
X
I
N
1

Figure 3—The output of the parse

to eliminate all backup. The statement recognizer identifies the type of each statement before the parse of that statement is attempted. The algorithm used by this procedure first attempts to recognize assignment statements using a left to right scan which looks for token patterns which are roughly analogous to $X =$ or $X\ (\ \ ) =$. If a statement is not recognized as an assignment, its leading token is matched against a keyword list to determine the statement type. This algorithm is very efficient and is able to positively identify all legal statements without requiring keywords to be reserved.

Declaration processing

PL/1 declaration processing is complicated by the great variety of data attributes and by the context sensitive manner in which they are derived. Two modules, the context processor and the declaration processor, process declarative information gathered by the parse.

The context processor

The context processor scans the data base containing contextually derived attributes produced during the parse by the context recorder. It either augments the partial symbol table created from declare statements or creates new declarations having the same format as those derived from declare statements. This activity creates contextual and implicit declarations.

The declaration processor

The declaration processor develops sufficient information about the variables of the program so that they may be allocated storage, initialized and accessed by the program's operators. It is organized to perform three major functions: the preparation of accessing code, the computation of each variable's storage requirements, and the creation of initialization code.

The declaration processor is relatively machine independent. All machine dependent characteristics, such as the number of bits per word and the alignment requirements of data types, are contained in a table. All computations or statements produced by the declaration processor have the same internal representation as source language expressions or statements. Later phases of the compiler do not distinguish between them.

The use of based references by the declaration processor

The concept of a based reference is useful to the understanding of PL/1 data accessing and the implementation of a number of language features. A based declaration of the form *DCL A BASED* is referenced by a based reference of the form $P \rightarrow A$, where P is a pointer to the storage occupied by a value whose description is given by the declaration of A. Multiple instances of data having the characteristics of A can be referenced through the use of unique pointers, i.e., $Q \rightarrow A$, $R \rightarrow A$, etc.

The declaration processor implements a number of language features by transforming them into suitable based declarations. Automatic data whose size is variable is transformed into a based declaration.

For example the declaration:

DCL A(N) AUTO;

becomes

DCL A(N) BASED(P);

where: P is a compiler produced pointer which is set upon entry to the declaring block.

Based declarations are also used to implement parameters. For example.

X: PROC (C); DCL C;

becomes

```
X: PROC (P); DCL C BASED(P);
```

where: *P* is a pointer which points to the argument corresponding to the parameter *C*.

Data accessing

The address of an item of PL/1 data consists of three basic parts: a pointer to some storage location, a word offset from that location and a bit offset from the word offset. Either or both offsets may be zero. The term "word" is understood to refer to the addressable unit of a computer's storage.

Example 1

```
DCL A AUTO;
```

The address of *A* consists of a pointer to the declaring block's automatic storage, a word offset within that automatic storage and a zero bit offset

Example 2

```
DCL 1 S BASED(P),
      2 A BIT(5),
      2 B BIT(N)
```

When referenced by $P \rightarrow B$, the address of *B* is a pointer *P*, a zero word offset and a bit offset of 5. The word offset may include the distance from the origin of the item's storage class, as was the case with the first example, or it may be only the distance from the level-one containing structure, as it was in the last example. The term "level-one" refers to all variables which are not contained within structures. Subscripted array element references, *A(K, J)*, or sub-string references, *SUBSTR(X, K, J)*, may also be expressed as offsets.

Offset expressions

The declaration processor constructs offset expressions which represent the distance between an element of a structure and the data origin of its level-one containing structure. If an offset expression contains only constant terms, it is evaluated by the declaration processor and results in a constant addressing offset. If the offset expression contains variable terms, the expression results in the generation of accessing instructions in the object program. The discussion which follows describes the efficient creation of these offset expressions.

Given a declaration of the form:

```
DCL 1 S,
      2 A BIT(M),
      2 B BIT(5),
      2 C FLOAT;
```

The offset of *A* is zero, the offset of *B* is *M* bits, and the offset of *C* is $M + 5$ bits rounded upward to the nearest word boundary.

In general, the offset of the nth item in a structure is:

$$b_n(c_{n-1}(s_{n-1}) + b_{n-1}(c_{n-2}(s_{n-2}) + b_{n-2} (\cdots b_3(c_2(s_2)) + b_2(c_1(s_1)))\cdots)))$$

where: b_k is a rounding function which expresses the boundary requirement of the *kth* item.

s_k is the size of the *kth* item.

c_k is the conversion factor necessary to convert s_k to some common units such as bits.

The declaration processor suppresses the creation of unnecessary conversion functions (c_k) and boundary functions (b_k) by keeping track of the current units and boundary as it builds the expression. As a result the offset expressions of the previous example do not contain conversion functions and boundary functions for *A* and *B*.

During the construction of the offset expression, the declaration processor separates the constant and variable terms so that the addition of constant terms is done by the compiler rather than by accessing code in the object program. The following example demonstrates the improvement gained by this technique.

```
DCL 1 S,
      2 A BIT(5),
      2 B BIT(K),
      2 C BIT(6),
      2 D BIT(10);
```

The offset of *D* is $K+11$ instead of $5+K+6$.

The word offset and the bit offset are developed separately. Within each offset, the constant and variable parts are separated. These separations result in the minimization of additions and unit conversions. If the declaration contains only constant sizes, the resulting offsets are constant. If the declaration contains expressions, then the offsets are expressions containing the minimum number of terms and conversion factors.

The development of size and offset expressions at compile time enables the object program to access data without the use of data descriptors or "dope vectors."[8] Most existing PL/1 implementations make extensive use of such descriptors to access data whose size or offsets are variable. Unless these descriptors are implemented by hardware, their use results in rather inefficient object code. The Multics PL/1 strategy of developing offset expressions from the declarations results in accessing code similar to that produced for subscripted array references. This code is generally more efficient than code which uses descriptors.

In general, the offset expressions constructed by the declaration processor remain unchanged until code generation. Two cases are exceptions to this rule: subscripted array references, $A(K,J)$, and sub-string references, $SUBSTR(X,K,J)$. Each subscripted reference or sub-string reference is a reference to a unique sub-datum within the declared datum and, therefore, requires a unique offset. The semantic translator constructs these unique offsets using the subscripts from the reference and the offset prepared by the declaration processor.

Allocation

The declaration processor does not allocate storage for most classes of data, but it does determine the amount of storage needed by each variable. Variables are allocated within some segment of storage by the code generator. Storage allocation is delayed because, during semantic translation and optimization, additional declarations of constants and compiler created variables are made.

Initialization

The declaration processor creates statements in the prologue of the declaring block which will initialize automatic data. It generates DO statements, IF statements and assignment statements to accomplish the required initialization.

The expansion of the initial attribute for based and controlled data is identical to that for automatic data except that the required statements are inserted into the program at the point of allocation rather than in the prologue.

Since array bounds and string sizes of static data are required by the language to be constant, and since all values of the initial attribute of static data must be constant, the compiler is able to initialize the static data at compile time. The initialization is done by the code generator at the time it allocates the static data.

Semantic translation

The semantic translator transforms the internal representation so that it reflects the attributes (semantics) of the declared variables without reflecting the properties of the object machine. It makes a single scan over the internal representation of the program. A compiler, which had no equivalent of the optimizer phase and which did not separate the machine dependencies into a separate phase, could conceivably produce object code during this scan.

Organization of the semantic translator

The semantic translator consists of a set of recursive procedures which walk through the program tree. The actions taken by these procedures are described by the general terms: operator transformation and operand processing. Operator transformation includes the creation of an explicit representation of each operator's result and the generation of conversion operators for those operands which require conversion. Operand processing determines the attributes, size and offsets of each operator's operands.

Operator transformation

The meaning of an operator is determined by the attributes of its operands. This meaning specifies which conversions must be performed on the operands, and it decides the attributes of the operator's result.

An operator's result is represented in the program tree by a temporary node. Temporary nodes are a further qualification of the original operator. For example, an add operator whose result is fixed-point is a distinct operation from an add operator whose result is floating-point. There is no storage associated with temporaries—they are allocated either core or register storage by the code generator. A temporary's size is a function of the operator's meaning and the sizes of the operator's operands. A temporary, representing the intermediate result of a string operation, requires an expression to represent its length if any of the string operator's operands have variable lengths.

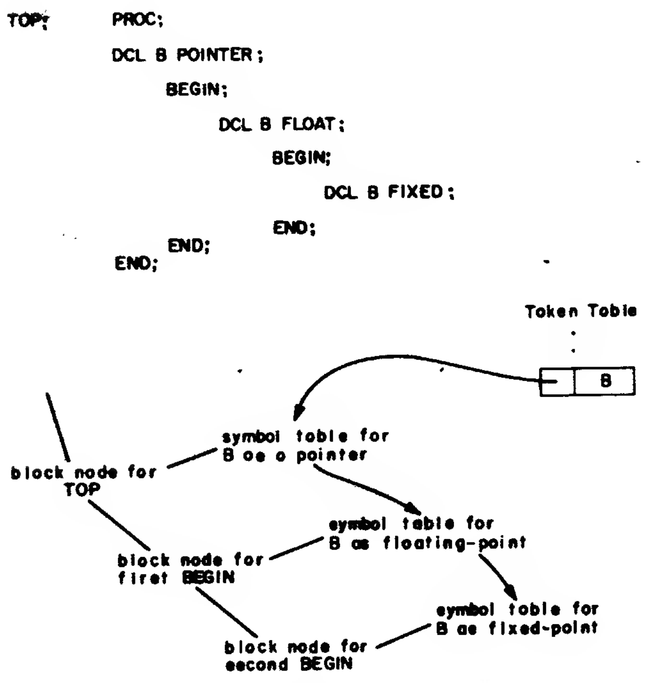
Operand processing

Operands consist of sub-expressions, references to variables, constants, and references to procedure names or built-in functions. Sub-expression operands are processed by recursive use of operator transformation and operand processing. Operand processing converts constants to a binary format which depends on the context in which the constant was used. References to variables or procedure names are associated with their appropriate declaration by the search function. After the search function has found the appropriate declaration, the reference may be further processed by the subscriptor or function processor.

The Search function

During the parse, it is not possible for references to source program variables to know the declared attributes of the variable because the PL/1 language allows declarations to follow their use. Therefore, references to source program variables are parsed into a form which contains a pointer to a token table entry rather than to a declaration of the variable. Figure 3 shows the output of the parse. The search function finds the proper declaration for each reference to a source program variable. The effectiveness of the search depends heavily on the structure of the token table and the symbol table. After declaration processing, the token table entry representing an identifier contains a list of all the declarations of that identifier. See Figure 4.

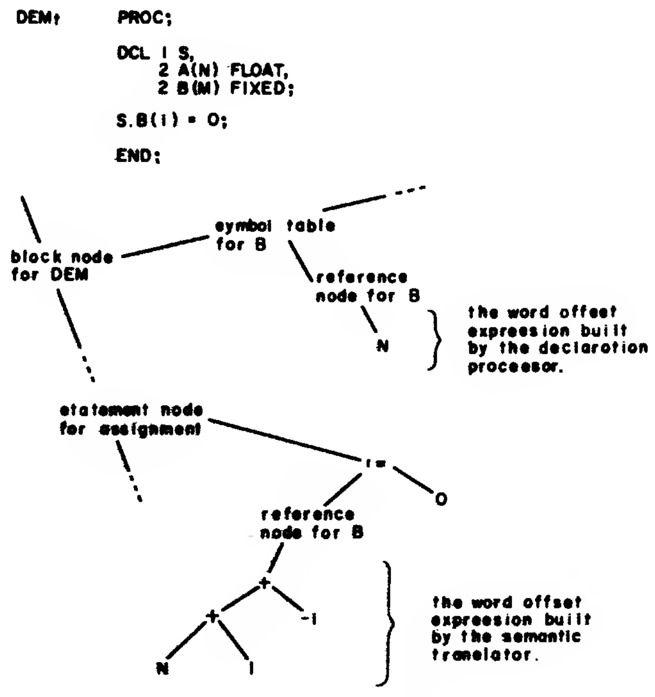
The search function first tries to find a declaration belonging to the block in which the reference occurred. If it fails to find one, it looks for a declaration in the next containing block. This process is repeated until a declaration is found. Since the number of declarations on the list is usually one, the search is quite fast. In its attempt to find the appropriate declaration, the search function obeys the language rules regarding structure qualification. It also collects any subscripts used in the reference and places them into a subscript list. Depending on the attributes of the referenced item, the subscript list serves as input to the function processor or subscriptor.



```
TOP:    PROC;
        DCL B POINTER;
            BEGIN;
                DCL B FLOAT;
                    BEGIN;
                        DCL B FIXED;
                    END;
            END;
        END;
```

Token Table
B
block node for TOP
symbol table for B as a pointer
block node for first BEGIN
symbol table for B as floating-point
block node for second BEGIN
symbol table for B as fixed-point

Figure 4—The relationship between the token table and the symbol table



```
DEM:    PROC;
        DCL 1 S,
            2 A(N) FLOAT,
            2 B(M) FIXED;
        S.B(I) = 0;
        END;
```

block node for DEM
symbol table for B
reference node for B
N
the word offset expression built by the declaration processor.
statement node for assignment
=
0
reference node for B
+
+
-1
N
I
the word offset expression built by the semantic translator.

Figure 5—A simplified diagram showing the effects of subscripting

The declaration processor creates offset expressions and size expressions for all variables. These expressions, known as accessing expressions, are rooted in a reference node which is attached to a symbol table node. The reference node contains all information necessary to access the data at run time. The search function translates a source reference into a pointer to this reference node. See Figure 5.

Subscripting

Since each subscripted reference is unique, its offset expression is unique. To reflect this in the internal representation, the subscriptor creates a unique reference node for each subscripted reference. See Figure 6. The following discussion shows the relationship between the declared array bounds, the element size, the array offset and subscripts.

Let us consider the case of an array declared:

$$a(l_1:u_1, l_2:u_2, \cdots, l_n:u_n)$$

Its element size is *s* and its offset is *b*.

The multipliers for the array are defined as:

$$\begin{aligned} m_n &= s \\ m_{n-1} &= (u_n - l_n + 1)s \\ m_{n-2} &= (u_{n-1} - l_{n-1} + 1)m_{n-1} \\ &\cdot \\ &\cdot \\ &\cdot \\ m_1 &= (u_2 - l_2 + 1)m_2 \end{aligned}$$

The offset of a reference $a(i_1, i_2, \cdots, i_n)$ is computed as:

$$v + \sum_{j=1}^{n} i_j m_j$$

where: *v* is the virtual origin. The virtual origin is the offset obtained by setting the subscripts equal to zero. It serves as a convenient base from which to compute the offset of any array element.

During the construction of all expressions, the constant terms are separated from the variable terms and all constant operations are performed by the compiler. Since the virtual origin and the multipliers are common to all references, they are constructed by the declaration processor and are repeatedly used by the subscriptor.

Arrays of PL/1 structures which contain arrays may result in a set of multipliers whose units differ. The declaration:

```
DCL 1 S(10),
      2 A PTR,
      2 B(10) BIT(2);
```

yields two multipliers of different units. The first multiplier is the size of an element of *S* in words, while the second multiplier is the size of an element of *B* in bits.

Array parameters which may correspond to an array cross section argument must receive their multipliers from an argument descriptor. Since the arrangement of the cross section elements in storage is not known to the called program, it cannot construct its own multipliers and must use multipliers prepared by the calling program. Note that the current definition of PL/1 allows any array parameter to receive a cross section argument.

```
FIG:    PROC;
        DCL (X,Y,Z) FLOAT;
        X = Y+Z;
        END;
```

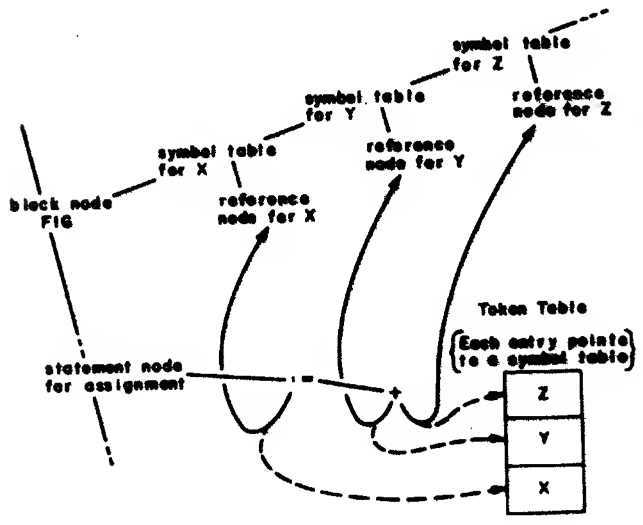


Figure 6—The internal representation of a statement before and after the execution of the search function. The broken lines show the statement's operands before the search

The function processor

An operand which is a reference to a procedure is expanded by the function processor into a call operator and possible conversion operators. Built-in function references result in new operators or are translated into expressions consisting of operators and operands.

Generic procedure references

A generic entry name represents a family of procedures whose members require different types of arguments.

```
DCL ALPHA GENERIC  (BETA
                    ENTRY(FIXED)),
                    GAMMA
                    ENTRY(FLOAT));
```

A reference to *ALPHA* (*X*) will result in a call to *BETA* or *GAMMA* depending on the attributes of *X*.

The declaration processor chains together all members of a generic family and the function processor selects the appropriate member of the family by matching the arguments used in the reference with the declared argument requirements of each member. When the appropriate member is found, the original reference is replaced by a reference to the selected member.

Argument processing

The function processor matches arguments to user-declared procedures against the argument types required for the procedure. It inserts conversion operators into the program tree where appropriate, and it issues diagnostics when it detects illegal cases.

The return value of a function is processed as if it were the n+1th argument to the procedure, eliminating the distinction between subroutines and functions.

The function processor determines which arguments may possibly correspond to a parameter whose size or array bounds are not specified in the called procedure. In this case, the argument list is augmented to include the missing size information. A more detailed description of this issue is given later in the discussion of object code strategies.

The built-in function processor

The built-in function processor is basically a table driven device. The driving table describes the number and kind of arguments required by each function and is used to force the necessary conversions and diagnostics for each argument. Most functions require processing which is unique to that function, but the table driven device minimizes the amount of this processing.

The *SUBSTR* built-in function is of particular importance since it is a basic PL/1 string operator. It is a three argument function which allows a reference to be made to a portion of a string variable, i.e., *SUBSTR* (*X*, *I*, *J*) is a reference to the ith through i + j − 1th character (or bit) in the string *X*.

This function is similar to an array element reference in the sense that they both determine the offsets of the reference. The processing of the *SUBSTR* function involves adjusting the offset and length expressions contained in the reference node of *X*. As is the case in all compiler operations on the offset expressions, the constant and variable terms are separated to minimize the object code necessary to access the data.

The optimizer

The compiler is designed to produce relatively fast object code without the aid of an optimizing phase. Normal execution of the compiler will by-pass the optimizer, but if extensively optimized object code is desired, the user may set a compiler command option which will execute the optimizer. The optimizer consists of a set of procedures which perform two major optimizations: common sub-expression removal and remova' of computations from loops. The data bases necessary for these optimizations are constructed by the parse and the semantic translator. These data bases consist of a cross-reference structure of statement labels and a tree structure representing the DO groups of each block. Both optimizations are done on a block basis using these two data bases.

Although the optimizer phase was not implemented at the time this paper was written, all data bases required by the optimizer are constructed by previous phases of the compiler and the abnormality of all variables is properly determined.

Optimization of PL/I programs

The on-condition mechanism of the PL/1 language makes the optimization of PL/1 programs considerably more difficult than the optimization of Fortran programs. Assuming that an optimized version of a program should yield results identical to those produced by the un-optimized version, then if any on-conditions are enabled in a given region of the program, the compiler cannot remove or reorder the computations performed in that region. (Consider the case of a divide by zero on unit which counts the number of times that the condition occurs.)

Since some on-conditions are enabled by default, most PL/1 programs cannot be optimized. Because of the difficulty of determining the abnormality of a program's variables, the optimization of those programs which may be optimized requires a rather intelligent compiler. A variable is abnormal in some block if its value can be altered without an explicit indication of that fact present in that block. An optimizing PL/1 compiler must consider all based variables, all arguments to the *ADDR* function, all defined variables, and all base items of defined variables to be abnormal. If the compiler expects values of variables to be retained throughout the execution of a call, it must also consider all parameters, all external variables, and all arguments of irreducible functions to be abnormal.

Because of the difficulty of optimizing programs written in the current PL/1 language[1] compilers should probably not attempt to perform general optimizations but should concentrate on special case optimizations which are unique to each implementation. Future revisions to the language definition may help solve the optimization problem.

The code generator

The code generator is the machine dependent portion of the compiler. It performs two major functions: it allocates data into Multics segments and it generates

645 machine instructions from the internal representation.

Storage allocation

A module of the code generator called the storage allocator scans the symbol table allocating stack storage for constant size automatic data, and linkage segment storage for internal static data. For each external name the storage allocator creates a link (an out-reference) or a definition (an entry point) in the linkage segment. All internal static data is initialized as its storage is allocated.

Due to the dynamic linking and loading characteristics of the Multics environment, the allocation and initialization of external static storage is rather unusual. The compiler creates a special type of link which causes the linker module of the operating system to create and initialize the external data upon first reference. Therefore, if two programs contain references to the same item of external data, the first one to reference that data will allocate and initialize it.

Code generation

The code generator scans the internal representation transforming it into 645 machine instructions which it outputs into the text segment. During this scan the code generator allocates storage for temporaries, and maintains a history of the contents of index registers to prevent excessive loading and storing of index values.

Code generation consists of three distinct activities: address computation, operator selection and macro expansion. Address computation is the process of transforming the offset expressions of a reference node into a machine address or an instruction sequence which leads to a machine address. Operator selection is the translation of operators into n-operand macros which reflect the properties of the 645 machine.

A one-to-one relationship often exists between the macros and 645 instructions but many operations (load long string, etc.) have no machine counterpart. All macros are expanded in actual 645 code by the macro expander which uses a code pattern table (macro skeletons) to select the specific instruction sequences for each macro.

Object code strategies

The object code design

The design of the object code is a compromise between the speed obtainable by straight in-line code and the necessity to minimize the number of page faults caused by large object programs.

The length of the object program is minimized by the extensive use of out-of-line code sequences. These out-of-line code sequences represent invariant code which is common to all Multics PL/1 object programs. Although the compiled code makes heavy use of out-of-line code sequences, the compiled code is not in any respect interpretive. The object code produce for each operator is very highly tailored to the specific attributes of that operator.

All out-of-line sequences are contained in a single "operator" segment which is shared by all users. The in-line code reaches on out-of-line sequence through transfer instructions, rather than through the standard subroutine mechanism. We believe that the time overhead associated with the transfers is more than redeemed by the reduction in the number of page faults caused by shorter object programs. System performance is improved by insuring that the pages of the operator segment are always retained in storage.

The stack

Multics PL/1 object programs utilize a stack segment for the allocation of all automatic data, temporaries, and data associated with on-conditions. Each task (Multics process) has its own stack which is extended (pushed) upon entry to block and is reverted (popped) upon return from a block. Prior to the execution of each statement it is extended to create sufficient space for any variable length string temporaries used in that statement. Constant size temporaries are allocated at compile time and do not cause the stack to be extended for each statement.

Prologue and epilogue

The term prologue describes the computations which are performed after block entry and prior to the execution of the first source statement. These actions include the establishment of the condition prefix, the computation of the size of variable size automatic data, extension of the stack to allocate automatic data, and the initialization of automatic data. Epilogues are not needed because all actions which must be undone upon exit from the block are accomplished by popping the stack. The stack is popped for each return or non-local go to statement.

Accessing of data

Multics PL/1 object code addresses all data, including members of variable sized structures and arrays directly through the use of in-line code. If the address of the data is constant, it is computed at compile time. If it is a mixture of constant and variable terms, the constant terms are combined at compile time. Descriptors are never used to address or allocate data.

String operations

All string operations are done by in-line code or by "transfer" type subroutinized code. No descriptors or calls are produced for string operations. The *SUBSTR* built-in function is implemented as a part of the normal addressing code and is therefore as efficient as a subscripted array reference.

String temporaries

A string temporary or dummy is designed in such a way that it appears to be both a varying and non-varying string. This means that the programmer does not need to be concerned with whether a string expression is varying or non-varying when he uses such an expression as an argument.

Varying strings

The Multics PL/1 implementation of varying strings uses a data format which consists of an integer followed by a non-varying string whose length is the declare maximum of the varying string. The integer is used to hold the current size of the string in bits or characters. Using this data format, operations on varying strings are just as efficient as operations on non-varying strings.

On-conditions

The design of the condition machinery minimizes the overhead associated with enabling and reverting on-units and transfers most of the cost to the signal statement. All data associated with on-conditions, including the condition prefix, is allocated in the stack. The normal popping of the stack reverts all enabled on-units and restores the proper condition prefix. Stack storage associated with each block is threaded backward to the previous block. The signal statement uses this thread to search back through the stack looking for the first enabled unit for the condition being signalled. Figure 7 shows the organization of enabled on-units in the stack.

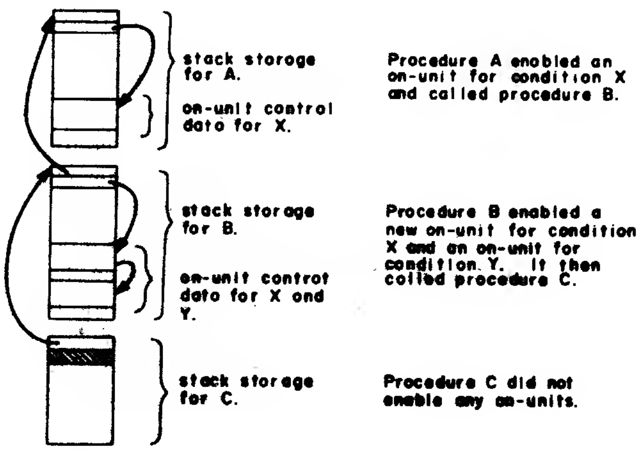

stack storage for A.
on-unit control data for X.
Procedure A enabled an on-unit for condition X and called procedure B.
stack storage for B.
on-unit control data for X and Y.
Procedure B enabled a new on-unit for condition X and an on-unit for condition. Y. It then called procedure C.
stack storage for C.
Procedure C did not enable any on-units.

Figure 7—Stack storage and the signal mechanism
A signal for condition X causes the signal mechanism to search back through the stack until it finds the first enabled on-unit for condition X.
An on-unit is compiled as an internal procedure. The execution of an ON-statement creates a block of on-unit control data. This control data consists of the name of the condition for which the unit was enabled and a procedure variable. The signal mechanism uses the procedure variable to invoke the on-unit. All data associated with the enabled on-unit is stored in the stack storage of the procedure which enabled it. Normal popping of the stack reverts the on-units enabled during the execution of the procedure.

Argument passing

The PL/1 language permits parameters to be declared with unknown array bounds or string lengths. In these cases, the missing size information is assumed to be supplied by the argument which corresponds to the parameter. This missing size information is not explicitly supplied by the programmer as is the case in Fortran, rather it must be supplied by the compiler as indicated in the following example:

```
SUB: PROC(A);        MAIN: PROC;
      .                    .
      .                    .
      .                    .
DCL A CHAR(*);       DCL SUB ENTRY;
      .
      .              DCL B CHAR(10);
      .
                          CALL SUB(B);
                               .
                               .
                               .
```

Since parameter *A* assumes the length of the argument *B*, the compiler must include the length of *B* in the argument list of the call to *SUB*.

The declaration of an entry name may or may not include a description of the arguments required by that entry. If such a description is not supplied, then the calling program must assume that argument descriptors are needed, and must include them in all calls to the entry. If a complete argument description is contained in the calling program, the compiler can determine if descriptors are needed for calls to the entry.

In the previous example the entry *SUB* was not fully declared and the compiler was forced to assume that an argument descriptor for *B* was required. If the entry had been declared *SUB ENTRY* (*CHAR*(*)) the compiler could have known that the descriptor of *B* was actually required by the procedure *SUB*. Since descriptors are often created by the calling procedure but not used by the called procedure, it is desirable to separate them from the argument information which is always used by the called procedure.

Communication between procedures written in PL/1 and other languages is facilitated if the other languages do not need to concern themselves with PL/1 argument descriptors. The Multics PL/1 implementation of the argument list is shown in Figure 8. Note that the argument pointers point directly to the data (facilitating communication between languages) and that the descriptors are optional, also note that PL/1 pointers must be capable of bit addressing in order to implement unaligned strings. Since descriptors contain no addressing information, they are quite often constant and can be prepared at compile time.

```
TAG:    PROC;
        DCL A(10) BIT(N),B CHAR(7),C AREA(1024);
        CALL X(A,B,C);
        END;
```

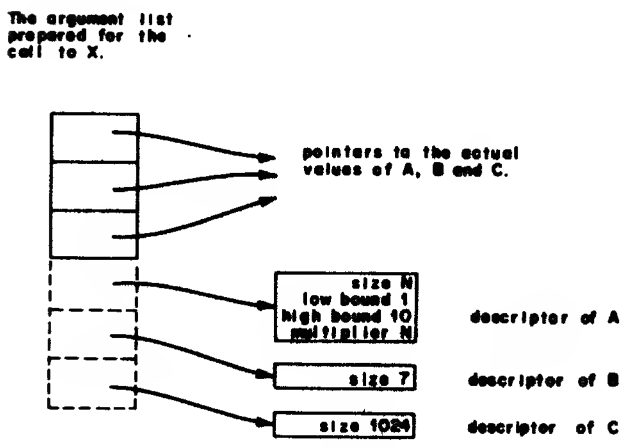


Figure 8—An argument list showing the relationship between arguments and their descriptors. The broken lines indicate that descriptors are optional.

SUMMARY

Our experiences both as users and implementors of PL/1 have led us to form a number of opinions and insights which may be of general interest.

1. It is feasible, but difficult, to produce efficient object code for the PL/1 language as it is currently defined. Unless a considerable amount of work is invested in a PL/1 compiler, the object code it generates will generally be much worse than that produced by most Fortran or COBOL compilers.
2. The difficulty of building a compiler for the current language has been seriously underestimated by most implementors. Unless the language is markedly improved and simplified this problem will continue to restrict the availability and acceptance of the language and will lead to the implementation of incompatible dialects and subsets.[7]
3. Simplification of the existing language will make it more suitable to users and implementors. We believe that the language can be simplified and still retain its "universal" character and capabilities.
4. The experience of writing the compiler in PL/1 convinced us that a subset of the language is well suited to system programming. This conviction is supported by Professor Corbato in his report on the use of PL/1 as an implementation language for the Multics system.[8] Many PL/1 concepts and constructs are valuable, but PL/1 structures and list processing seem to be the principal improvement over alternative languages.[9]

ACKNOWLEDGMENTS

The author wishes to express recognition to members of the General Electric Multics PL/1 Project for their contributions to the design and implementation of the compiler. J. D. Mills was responsible for the design and implementation of the syntactic analyzer and the Multics system interface, B. L. Wolman designed and built the code generator and operator segment, and G. D. Chang implemented the semantic translator. Valuable advice and ideas were provided by A. H. Kvilekval. The earlier work of M. D. McIlroy and R. Morris of Bell Telephone Laboratories and numerous

persons at MIT's Project MAC provided a useful guide and foundation for our efforts.

REFERENCES

1 *PL/1 language specifications*
Form Y33-6003-0 IBM Corp March 1968

2 The formal definition of PL/1 as specified by technical reports TR25.081, TR25.082, TR25.083, TR25.084, TR25.085, TR25.086 and TR25.087, IBM Corp Vienna Austria June 1968

3 F J CORBATO V A VYSSOTSKY
Introduction and overview of the multics system
Proc FJCC 1965

4 V A VYSSOTSKY F J CORBATO R M GRAHAM
Structure of the multics supervisor
Proc FJCC 1965

5 R C DALEY J B DENNIS
Virtual memory, processes, and sharing in multics
CACM Vol 11 No 5 May 1968

6 *PL/1 (F) programmer's guide*
Form C28-6594-3 IBM Corp Oct 1967

7 R F ROSIN
PL/1 Implementation survey
ACM SIGPLAN Notices Feb 1969

8 F J CORBATO
PL/1 as a tool for system programming
Datamation May 1969

9 H W LAWSON JR
PL/1 list processing
CACM Vol 10 No 6 June 1967

Remote Terminal Character Stream Processing in Multics

by J.H. Saltzer and J.F. Ossanna. Reprinted from AFIPS Conference Proceedings 36, AFIPS Press, 1970, pp. 621-627, with permission. Copyright 1970 by AFIPS Press.

This paper describes one of the numerous areas of an operating system which must be carefully thought out to provide a uniform, well-engineered human interface. The topic is the processing of terminal input and output so that programs see a standard implementation-independent terminal, while typists see a simple, easy-to-learn method of communicating with the system, no matter which terminal device they happen to be faced with. Since the system has been used with perhaps 25 different kinds of terminal equipment the considerations described here cannot be ignored. (Note, however, that we are here dealing with a set of concepts which are a notch below the importance of, say, the Multics virtual memory strategy.) The paper is generally up-to-date in terminology, but for exact details of the typing conventions one should refer to section 1 of the Reference Guide.

Reprinted from –
AFIPS – Conference Proceedings, Volume 36
Copyright © by AFIPS Press
Montvale, New Jersey
07645

Remote terminal character stream processing in Multics

by J. H. SALTZER
Massachusetts Institute of Technology
Cambridge, Massachusetts

and

J. F. OSSANNA
Bell Telephone Laboratories, Inc.
Murray Hill, New Jersey

INTRODUCTION

There are a variety of considerations which are pertinent to the design of the interface between programs and typewriter-class remote terminal devices in a general-purpose time-sharing system. The conventions used for editing, converting, and reduction to canonical form of the stream of characters passing to and from remote terminals is the subject of this paper. The particular techniques used in the Multics* system are presented as an example of a single unified design of the entire character stream processing interface. The sections which follow contain discussion of character set considerations, character stream processing objectives, character stream reduction to canonical form, line and print position deletion, and other interface problems. An appendix gives a formal description of the canonical form for stored character strings used in Multics.

CHARACTER SET CONSIDERATIONS

Although for many years computer specialists have been willing to accept whatever miscellaneous collection of characters and codes their systems were delivered with, and to invent ingenious compromises when designing the syntax of programming languages, the impact of today's computer system is felt far beyond the specialist, and computer printout is (or should be) received by many who have neither time nor patience to decode information printed with inadequate graphic versatility. Report generation has, for some time, been a routine function. Recently, on-line documentation aids, such as RUNOFF,[3] Datatext (IBM Corp.) or RAES (General Electric Co.) have attracted many users. Especially for the latter examples it is essential to have a character set encompassing both upper and lower case letters. Modern programming languages can certainly benefit from availability of a variety of special characters as syntactic delimiters, the ingenuity of PL/I in using a small set notwithstanding.

Probably the minimum character set acceptable today is one like the USASCII 128-character set[4] or IBM's EBCDIC set with the provision that they be fully supported by upper/lower case printer and terminal hardware. The definition of support of a character set is almost as important as the fact of support. To be fully useful, one should be able to use the same full character set in composing program or data files, in literal character strings of a programming language, in arguments of calls to the supervisor and to library routines requiring symbolic names, as embedded character strings in program linkage information, and in input and output to typewriters, displays, printers, and cards. However, it may be necessary to place conversion packages in the path to and from some devices since it is rare to find that all the different hardware devices attached to a system use the same character set and character codes.

* Multics is a comprehensive general purpose time-sharing system implemented on the General Electric 645 computer system. A general description of Multics can be found in Reference 1 or 2.

TABLE I—Escape conventions for input and output of USASCII from an EBCDIC typewriter

ASCII Character Name	ASCII Graphic	Normal EBCDIC Escape	Alternate "edited" Escape
Right Square Bracket	]	¢>	≱
Left Square Bracket	[	¢<	≰
Right Brace	}	¢)	-)
Left Brace	{	¢(	-(
Tilde	~	¢t	-(
Grave Accent		¢'	

CHARACTER STREAM PROCESSING CONSIDERATIONS

The treatment of character stream input and output may be degraded, from a human engineering point of view, unless it is tempered by the following two considerations:

1. If a computer system supports a variety of terminal devices (Multics, for example, supports both the IBM Model 2741* and the Teletype Model 37*) then it should be possible to work with any program from any terminal.
2. It should be possible to determine from the printed page, without ambiguity, both what went into the computer program and what the program tried to print out.

To be fully effective, these two considerations must apply to all input and output to the system itself (e.g., when logging in, choosing subsystems, etc.) as well as input and output from user programs, editors, etc.

As an example of the "device independence" convention, Multics uses the USASCII character set in all internal interfaces and provides standard techniques for dealing with devices which are non-USASCII. When using the GE-645 USASCII line printer or the Teletype Model 37, there is no difficulty in accepting any USASCII graphic for input or output from any user or system program. In order to use non-USASCII hardware devices, one USASCII graphic (the left slant) is set aside as a "software escape" character. When a non-USASCII device (say the IBM 2741 typewriter with an EBCDIC print element) is to be used, one first makes a correspondence, so far as possible, between graphics available on the device and graphics of USASCII, being sure that some character of the device corresponds to the software escape character. Thus, for the IBM 2741, there are 85 obviously corresponding graphics; the EBCDIC overbar, cent sign, and apostrophe can correspond to the USASCII circumflex, left slant, and acute accent respectively, leaving the IBM 2741 unable to represent six USASCII graphic characters. For each of the six missing characters a two character sequence beginning with the software escape character is defined, as shown in Table I. The escape character itself, as well as any illegal character code value, is represented by a four character sequence, namely the escape character followed by a 3-digit octal representation of the character code. Thus, it is possible from an IBM 2741 to easily communicate all the characters in the full USASCII set.

A similar, though much more painful, set of escape conventions has been devised for use of the Model 33 and 35 Teletypes. The absence of upper and lower case distinction on these machines is the principal obstacle; two printed 2-character escape sequences are used to indicate that succeeding letters are to be interpreted in a specific case shift.

Note that consideration number two above, that the printed record be unambiguous, militates against character set extension conventions based on non-printing and otherwise unused control characters. Such conventions inevitably lead to difficulty in debugging, since the printed record cannot be used as a guide to the way in which the input was interpreted.

The objective of typewriter device independence also has some implications for control characters. The Multics strategy here is to choose a small subset of the possible control characters, give them precise meanings, and attempt to honor those meanings on every device, by interpretation if necessary. Thus, a "new page" character appears in the subset; on a Model 37 teletype it is interpreted by issuing a form feed and a carriage return; on an IBM 2741 it is interpreted by giving an appropriate number of new line characters.*

Of the 33 possible USASCII control characters, 11 are defined in Multics as shown in Table II.

Red and black shift characters appear in the set because of their convenience in providing emphasis in comments, both by system and by user routines. The half-line forward and half-line reverse feed characters were included to facilitate experimentation with the Model 37 Teletype; these characters are not currently interpretable on other devices.

One interesting point is the choice of a "null" or "padding" character used to fill out strings after the last meaningful character. By convention, padding characters appearing in an output stream are to be discarded, either by hardware or software. The USASCII choice of code value *zero* for the null character has the

*This interpretation of the form feed function is consistent with the International Standards Organization option of interpreting the "line feed" code as "new line" including carriage return.

interesting side effect that if an uninitialized string (or random storage area) is unintentionally added to the output stream, all of the zeros found there will be assumed nulls, and discarded, possibly leaving no effect at all on the output stream. Debugging a program in such a situation can be extraordinarily awkward, since there is no visible evidence that the code manipulating the offending string was ever encountered.

In Multics, this problem was considered serious enough that the USASCII character "delete" (all bits *one*) was chosen as the padding character code. The *zero* code is considered illegal, along with all other unassigned code values, and is printed in octal whenever encountered.

As an example of a control function not appearing in the character set, the printer-on/printer-off function (to allow typing of passwords) is controlled by a special call which must be inserted before the next call to read information. This choice is dictated by the need to get back a status report which indicates that for the currently attached device, the printer cannot be turned on and off. Such a status report can be returned as an error code on a special call; there would be no convenient way to return such status if the function were controlled by a character in the output stream.**

TABLE II—USASCII Control Characters as Used in Multics

USASCII NAME	MULTICS NAME	MULTICS MEANING
BEL	BEL	Sound an audible alarm.
BS	BS	Backspace. Move carriage back one column. The backspace implies over-striking rather than erasure.
HT	HT	Horizontal Tabulate. Move carriage to next horizontal tab stop. Default tab stops are assumed to be at columns 11, 21, 31, 41, etc.
LF	NL	New Line. Move carriage to left edge of next line.
SO	RRS	Red Ribbon Shift.
SI	BRS	Black Ribbon Shift.
VT	VT	Vertical Tabulate. Move carriage to next vertical tab stop. Default tab stops are assumed to be at lines 11, 21, 31, etc.
FF	NP	New Page. Move carriage to the left edge of the top of the next page.
DC2	HLF	Half-Line Forward Feed.
DC4	HLR	Half-Line Reverse Feed.
DEL	PAD	Padding Character. This character is discarded when encountered in an output line.

CANONICAL FORM FOR STORED CHARACTER STRINGS

Probably the most significant impact of the constraint that the printed record be unambiguous is the interaction of that constraint with the carriage motion control characters of the USASCII and EBCDIC sets. Although most characters imply "type a character in the current position and move to the next one," three commonly provided characters, namely backspace, horizontal tab, and carriage return (no line feed) do cause ambiguity.

For example, suppose that one chooses to implement an ALGOL language in which keywords are underlined. The keyword $\underline{\text{for}}$ may now be typed in at least a dozen different ways, all with the same printed result but all with different orders for the individual letters and backspaces. It is unreasonable to expect a translator to accept a dozen different, but equivalent, ways of typing every control word; it is equally unreasonable to require that the typist do his underlining in a standard way since if he slips, there is no way he can tell from his printed record (or later protestations of the compiler) what he has done wrong. A similar dilemma occurs in a manuscript editing system if the user types in underlined words, and later tries to edit them.

An answer to this dilemma is to process all character text entering the system to convert it into a *canonical form*. For example, on a "read" call Multics would return the string:

$$\_\langle BS\rangle f\_\langle BS\rangle o\_\langle BS\rangle r$$

(where $\langle BS\rangle$ is the backspace character) as the canonical character string representation of the printed image of $\underline{\text{for}}$ independently of the way in which it had been typed. Canonical reduction is accomplished by scanning across a completed input line, associating a carriage position with each printed graphic encountered, then sorting the graphics into order by carriage or print position. When two or more graphics are found in the same print position, they are placed in order by numerical collating sequence with backspace characters between. Thus, if two different streams of characters produce the same printed image, after canonical reduction they will be represented by the same stored string. Any program can thus easily compare two canonical strings to discover if they produce the same printed image. No restriction is

** The initial Multics implementation temporarily uses the character codes for USASCII ACK and NAK for this purpose, as an implementation expedient. In addition, a number of additional codes are accepted to permit experimentation with special features of the Model 37 Teletype; these codes may become standard if the features they trigger appear useful enough to simulate on all devices.

placed on the human being at his console; he is free to type a non-canonical character stream. This stream will automatically be converted to the canonical form before it reaches his program. (There is also an escape hatch for the user who wants his program to receive the raw input from his typewriter, unprocessed in any way.)

Similarly, a typewriter control module is free to rework a canonical stream for output into a different form if, for example, the different form happens to print more rapidly or reliably.

In order to accomplish canonical reduction, it is necessary that the typewriter control module be able to determine unambiguously what precise physical motion of the device corresponds to the character stream coming from or going to it. In particular, it must know the location of physical tab settings. This requirement places a constraint on devices with movable tab stops; when the tab stops are moved, the system must be informed of the new settings.

The apparent complexity of the Multics canonical form, which is formally described in Appendix I, is a result of its generality in dealing with all possible combinations of typewriter carriage motions. Viewed in the perspective of present day language input to computer systems, one may observe that many of the alternatives are rarely, if ever, encountered. In fact for most input, the following three statements, describing a simplified canonical form, are completely adequate:

1. A message consists of strings of character positions separated by carriage motion.
2. Carriage motions consist of New Line or Space characters.
3. Character positions consist of a single graphic or an overstruck graphic. A character position representing overstrikes contains a graphic, a backspace character, a graphic, etc., with the graphics in ascending collating sequence.

Thus we may conclude that for the most part, the canonical stream will differ little with the raw input stream from which it was derived.

A strict application of the canonical form as given in Appendix I has a side effect which has affected its use in Multics. Correct application leads to replacement of all horizontal tab characters with space characters in appropriate numbers. If one is creating a file of tabular information, it is possible that the ambiguity introduced by the horizontal tab character is in fact desirable; if a short entry at the left of a line is later expanded, words in that entry move over, but items in columns to the right of that entry should stay in their original carriage position; the horizontal tab facilitates expressing this concept. A similar comment applies to the form feed character.

The initial Multics implementation allows the horizontal tab character, if typed, to sneak through the canonical reduction process and appear in a stored string. A more elegant approach to this problem is to devise a set of conventions for a text editor which allows one to type in and edit tabular columns conveniently, even though the information is stored in strictly canonical form. Since the most common way of storing a symbolic program is in tabular columns, the need for simple conventions to handle this situation cannot be ignored.

It is interesting to note that most format statement interpreters, such as those commonly implemented for FORTRAN and PL/I, fail to maintain proper column alignment when handed character strings containing embedded backspaces, such as names containing overstruck accents. For complete integration of such character strings into a system, one should expand the notion of character counts to include print position counts as well as storage position counts. For example, the value returned by a built-in string length function should be a print position count if the result is used in formatting output; it should be a storage location count if the result is used to allocate space in memory.

LINE AND PRINT POSITION DELETION CONVENTIONS

Experience has shown that even with sophisticated editor programs available, two minimal editing conventions are very useful for human input to a computer system. These two conventions give the typist these editing capabilities at the instant he is typing:

1. Ability to delete the last character or characters typed.
2. Ability to delete all of the current line typed up to the point.

(More complex editing capabilities must also be available, but they fall in the domain of editing programs which can work with lines previously typed as well as the current input stream.) By framing these two editing conventions in the language of the canonical form, it is possible to preserve the ability to interpret unambiguously a typed line image despite the fact that editing was required.

The first editing convention is to reserve one graphic, (in Multics, the "number" sign), as the *erase* character. When this character appears in a print position, it erases itself and the contents of the previous print position. If the erase follows simple carriage motion, the entire carriage motion is erased. Several successive erase characters will erase an equal number of preceding print positions or simple carriage motions. Since erase processing occurs after the transformation to canonical form, there is no ambiguity as to which print position is erased; the printed line image is always the guide. Whenever a print position is erased, the carriage motions (if any) on the two sides of the erased print position are combined into a single carriage motion.

The second editing convention reserves another graphic (in Multics, the "commercial at" sign) as the *kill* character. When this character appears in a print position, the contents of that line up to and including the kill character are discarded. Again, since the kill processing occurs after the conversion to canonical form, there can be no ambiguity about which characters have been discarded. By convention, kill is done before erase, so that it is not possible to erase a kill character.

OTHER INTERFACE CONVENTIONS

Two other conventions which can smooth the human interface on character stream input and output are worth noting. The first is that many devices contain special control features such as line feed without carriage movement, which can be used to speed up printing in special cases. If the system-supplied terminal control software automatically does whatever speedups it can identify, the user is not motivated to try to do them himself and risk dependence on the particular control feature of the terminal he happens to be working with. For example, the system can automatically insert tabs (followed by backspaces if necessary) in place of long strings of spaces, and it also can type centered short tabular information with line feed and backspace sequences between lines.

The second convention has been alluded to already. If character string input is highly processed for routine use, there must be available an escape by which a program can obtain the raw, unconverted, unreduced and unedited keystrokes of the typist, if it wants to. Only through such an escape can certain special situations (including experimenting with a different set of proposed processing conventions) be handled. In Multics, there are three modes of character handling—normal, raw, and edited.* The raw mode means no processing whatsoever on input or output streams, while the normal mode provides character escapes, canonical reduction, and erase and kill editing. The edited mode (effective only on output if requested) is designed to produce high quality clean copy; every effort is made to avoid using escape conventions. For example, illegal characters are discarded and graphics not available on the output device used are typed with the "overstrike" escapes of Table I, or else left as a blank space so that they may be drawn in by hand.

CONCLUSIONS

The preceding sections have discussed both the background considerations and the design of the Multics remote terminal character stream interface. Several years of experience in using this interface, first in a special editor on the 7094 Compatible Time-Sharing System and more recently as the standard system interface for Multics, have indicated that the design is implementable, usable and effective. Probably the most important aspect of the design is that the casual user, who has not yet encountered a problem for which canonical reduction, or character set escapes, or special character definitions are needed, does not need to concern himself with these ideas; yet as he expands his programming objectives to the point where he encounters one of these needs, he finds that a method has been latently available all along in the standard system interface.

There should be no assumption that the particular set of conventions described here is the only useful set. At the very least, there are issues of taste and opinion which have influenced the design. More importantly, systems with only slightly different objectives may be able to utilize substantially different approaches to handling character streams.

ACKNOWLEDGMENTS

Many of the techniques described here were developed over a several year time span by the builders of the 7094 Compatible Time-Sharing System (CTSS) at MIT Project MAC, and by the implementers of Multics, a cooperative research project of the General Electric Company, the Bell Telephone Laboratories, Inc., and the Massachusetts Institute of Technology.

The usefulness of a canonical form for stored character strings was independently brought to our attention by E. Van Horne and C. Strachey; they had each implemented simple canonical forms on CTSS and in the TITAN operating system for the ATLAS computer, respectively. F. J. Corbató and R. Morris developed the pattern of escape sequence usage described here. Others contributing to the understanding of the issues involved in the character stream interface were R. C. Daley, S. D. Dunten, and M. D. McIlroy.

Work reported here was supported in part by the advanced Research Projects Agency, Department of

*The "raw" mode is not yet implemented.

Defense, under Office of Naval Research Contract Nonr-4102(01). Reproduction is permitted for any purpose of the United States Government.

REFERENCES

1 F J CORBATÓ et al
A new remote-accessed man-machine system
AFIPS Conference Proceedings 27 1965 FJCC Spartan Books Washington D C 1965 pp 185–247

2 *The multiplexed information and computing service: Programmer's manual*
M I T Project MAC Cambridge Massachusetts 1969 To be published

3 J H SALTZER
Manuscript typing and editing
In The Compatible Time-Sharing Sytem: A Programmer's Guide 2nd Edition M I T Press Cambridge Massachusetts 1965

4 *USA standard code for information interchange*
X3 4-1968 USA Standards Institute October 1968

5 *IBM 2741 communications terminal*
IBM Systems Reference Library Form A24-3415 IBM Corporation New York

6 *Model 37 teletypewriter stations for DATA-PHONE service*
Bell System Data Communications Technical Reference American Telephone and Telegraph Company New York September 1968

7 *PL/I language specifications*
IBM System Reference Library Form C28-6571 IBM Corporation New York

APPENDIX I

The Multics canonical form

To describe the Multics canonical form, we give a set of definitions of a canonical message. Each definition is followed by a discussion of its implications. PL/I-style formal definitions are included for the benefit of readers who find them useful.[7] Other readers may safely ignore them at a small cost in precision. In the formal definitions, capitalized abbreviations stand for the control characters in Table II.

1. The canonical form deals with messages. A message consists of a sequence of print positions, possibly separated by, beginning, or ending with carriage motion.

message : : = [carriage motion]
[[print position]···[carriage motion]]···

Typewriter input is usually delimited by *action* characters, that is, some character which, upon receipt by the system, indicates that the typist is satisfied with the previous string of typing. Most commonly, the new line character, or some variant, is used for this function. Receipt of the action character initiates canonical reduction.

The most important property on the canonical form is that graphics are in the order that they appear on the printed page reading from left to right and top to bottom. Between the graphic characters appear only the carriage motion characters which are necessary to move the carriage from one graphic to the next. Overstruck graphics are stored in a standard form including a backspace character (see below).

2. There are two mutually exclusive types of carriage motion, gross motion and simple motion.

$$\text{carriage motion} ::= \left\{\begin{array}{l}\text{gross motion}\\ \text{simple motion}\\ \text{gross motion simple motion}\end{array}\right\}$$

Carriage motion generally appears between two graphics; the amount of motion represented depends only on the relative position of the two graphics on the page. Simple motion separates characters within a printed line; it includes positioning, for example, for superscripts and subscripts. Gross motion separates lines.

3. Gross motion consists of any number of successive New Line (NL) characters.

$$\text{gross motion} ::= \{\text{NL}\}\cdots$$

The system must translate vertical tabs and form feeds into new line characters on input.

4. Simple motion consists of any number of Space characters (SP) followed by some number (possibly zero) of vertical half-line forward (HLF) or reverse (HLR) characters. The number of vertical half line feed characters is exactly the number needed to move the carriage from the lowest character of the preceding print position to the highest character of the next print position.

$$\text{simple motion} ::= \{\text{SP}\}\cdots\begin{bmatrix}[\text{HLF}]\cdots\\ \\ [\text{HLR}]\cdots\end{bmatrix}$$

The basis for the amount of simple carriage motion represented is always the horizontal and vertical distance between successive graphics that appears on the actual device. In the translation to and from the canonical form, the system must of course take into account the actual (possibly variable) horizontal tab stops on the physical device.

In some systems, a "relative horizontal tab" character is defined. Some character code (for example, USASCII DC1) is reserved for this meaning, and by convention the immediately following character storage position contains a count which is interpreted as the size of the horizontal white space to be left. Such a character fits smoothly into the canonical form described here in place of the successive spaces implied by the definition above. It also minimizes the space requirement of a canonical string. It does require some language features, or subroutines, to extract the count as an integer, to determine its size. It also means that character comparison is harder to implement; equality of a character with one found in a string may mean either that the hoped for character has been found or it may mean that a relative tab count happens to have the same bit pattern as the desired character; reference to the previous character in the string is required to distinguish the two cases.

5. A print position consists of some non-zero number of character positions, occupying different half line vertical positions in the same horizontal carriage position. All but the last character position of a print position are followed by a backspace character and some number of HLF characters.

print position : : = character position
[BS [HLF]···character position]···

6. A character position consists of a sequence of graphic formers separated by backspace characters. The graphic formers are ordered according to the USASCII coded numeric value of the graphics they contain. (The first graphic former contains the graphic with the smallest code, etc.) Two graphic formers containing the same graphic will never appear in the same character position.

character position : : = graphic former
[BS graphic former]···

Note that all possible uses of a backspace character in a raw input stream have been covered by statements about horizontal carriage movements and overstruck graphics.

7. A graphic former is a possibly zero-length setup sequence of graphic controls followed by one of the 94 USASCII non-blank graphic characters.

$$\text{graphic former} \; : \; : \; = \; [\text{setup sequence}] \left\{ \begin{matrix} \text{one of the} \\ \text{94 UASCII} \\ \text{graphic} \\ \text{characters} \end{matrix} \right\}$$

8. A graphic setup sequence is a color shift or a bell (BEL) or a color shift followed by a bell. The color shift only appears when the following graphic is to be a different color from the preceding one in the message.

$$\text{setup sequence} \; : \; : \; = \left\{ \begin{matrix} \left[\begin{matrix} \text{RRS} & \\ & [\text{BEL}] \\ \text{BRS} & \end{matrix} \right] \\ \\ \text{BEL} \end{matrix} \right\}$$

in the absence of a color shift, the first graphic in a message is printed in black shift. Other control characters are treated similarly to bell. They appear immediately before the next graphic typed, in the order typed.

By virtue of the above definitions, the control characters HT, VT, and CR will never appear in a canonical stream.

The Multics Input/Output System

by R.J. Feiertag and E.I. Organick. Reprinted from ACM Third Symposium on Operating System Principles, Palo Alto, California, October, 1971, pp. 35-41, with permission.

This generally up-to-date paper describes the device-independent I/O interface of the Multics system. Its significance lies mainly in the wide range of problems which can be easily solved using a simple elegantly designed mechanism.

By reading between the lines, one may also deduce that in Multics, the function of the I/O system is drastically different from that in most operating systems. Interrupt handling, scheduling, and file formatting do not appear here, since they are considered to be general responsibilities required apart from I/O operations. The I/O system is thus left with only the problem of buffer management and device strategy, in a general framework which encourages device independence.

As an example of the flexibility of the Multics I/O system, since this paper was written the M.I.T. Multics site has been attached to the ARPA computer network, with the relatively minor addition of a special network demultiplexing module at the base of the I/O system.

THE MULTICS INPUT/OUTPUT SYSTEM*

R. J. Feiertag
Massachusetts Institute of Technology
Cambridge, Massachusetts

and

E. I. Organick
University of Utah
Salt Lake City, Utah

ABSTRACT

An I/O system has been implemented in the Multics system that facilitates dynamic switching of I/O devices. This switching is accomplished by providing a general interface for all I/O devices that allows all equivalent operations on different devices to be expressed in the same way. Also particular devices are referenced by symbolic names and the binding of names to devices can be dynamically modified. Available I/O operations range from a set of basic I/O calls that require almost no knowledge of the I/O System or the I/O device being used to fully general calls that permit one to take full advantage of all features of an I/O device but require considerable knowledge of the I/O System and the device. The I/O System is described and some popular applications of it, illustrating these features, are presented.

Introduction

In many early operating system designs the software known as the input/output control system (IOCS) played a central conceptual and functional role. In the pre-multiprogramming, batch operating systems, many supervisory functions had to do with input/output control -- e.g., control over queued jobs, control for management and operation of secondary storage, control for operation of display devices and other peripheral equipment, etc. A system programmer (or subsystem designer) for such operating systems could hardly prove his professional competence without acquiring a reasonable familiarity with the intricacies of the IOCS for his "installation". By contrast the role played by the input/output control system in a Multics system is decidedly secondary, at least from a conceptual point of view. In fact, many or even most subsystem designers are able to achieve their respective objectives while remaining entirely oblivious to the IOCS details of Multics.

This is possible partly because two operations sometimes associated with the IOCS have been separated into separate functional units which are made use of by other parts of the system as well as the IOCS. First, the file system [1] makes known and dynamically links files that are stored within the system to processes that legitimately request this service. It does not matter on what storage device these files reside at the time of the request. The users (or for that matter other supervisory modules) are unaware of any explicit data movement in accessing these segments even though physical transfer from actual secondary devices to central memory may occur. Secondly, the traffic controller [2] handles all multiplexing of processors including the relinquishing of a processor by a process and the awakening of processes which have been waiting for I/O transactions to be completed. What remains for the IOCS is strategic control of I/O devices and the binding of these devices with symbolic names used to represent them. Figure 1 illustrates the interrelationships of these modules.

*Work reported herein was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract N0014-70-A-0362-0001. Reproduction is permitted for any purpose of the United States Government.

The secondary role of the I/O System does not mean that Multics attempts to erect a barrier that prevents the (system or user) programmer from acquiring and exercising full control over I/O devices. On the contrary, user processes are able to "negotiate" with the system administrator, who controls distribution of I/O resources, to acquire particular I/O devices. Then, with user code, the user process may program the control of these I/O devices and operate them with the full freedom that is normally accorded a system programmer.

In brief, the Multics I/O System has been designed using two important guidelines:

a) the simplest, most commonplace use of it requires only a minimum of knowledge and skill -- and the overhead for such simple (common mode) use is also minimized.

b) to extract more tailored (special purpose) services there is added cost -- both in the time that must be committed to understand how the tool works and in the actual overhead that will be incurred in execution.

The system to be described here stresses symbolic, hardware independent references to input/output devices. This scheme permits programs to be written largely independently of the devices they use and allows the devices to be assigned at the time the computation is performed and changed dynamically during the run. Although other systems [3,4,5] have made use of symbolic referencing, the Multics system attempts to provide extreme ease of modification and almost total device independence, to the limits possible.

The I/O System does not in itself provide formatted I/O such as that typically found in many languages and library subroutines. Also, the details of operating specific devices are relegated to a minor role. What remains is an intermediate level of I/O software that forms the conceptual heart of the I/O System in Multics and will now be described.

Overview of the I/O System

A primary objective of Multics is to make the input/output operations stated in the programs or service procedures that a user writes specify only those device functions that are required for the application at hand, leaving to the system the responsibility for gauging the degree of device independence implied by the user's request. In this way a user who invokes such service procedures is free to designate substitute devices as may be appropriate, while adhering to the device dependencies that are implied by the stated I/O function requests. For example, a program may output a long string of characters. If the device currently associated with this output is a typewriter the I/O System should insert carriage returns when the end of the carriage is reached. However, if the output device is a tape then no carriage returns are necessary. For this reason user-coded I/O operations should ordinarily be independent (or as independent as feasible) of the particular device and model, or even of the type of device, e.g., typewriter, as opposed to teletype or paper tape.

There are two clear motivations for this crucially important objective. First, we must presume that at any given time a system will generally accommodate several types of I/O devices and models. Each is likely to require different programmed control. Each may have different character sets, and may be intrinsically different in various respects (e.g., line printers are not backspaceable, magnetic tapes are; some tapes cannot be read backwards as well as forwards, while card readers are never designed to read cards backwards, etc.). It is, however, desirable to be able to run programs using devices other than those for which they were originally written. Second, we presume that I/O devices become obsolete and, over time, are replaced by new models of the same or different types, e.g., video keyboards may replace typewriters. Clearly, if programs are to be usable over long periods of time, if programs are to be repeated with minor or no variation in the nature or effect of their I/O operations, then recognition of device independence must be a planned part of the programming system for I/O operations.

One approach to design for the needed device independence is to regard the I/O resource needed to complete any given I/O operation not as a real or physical resource, as for instance a particular card reader, but as a virtual (paeudo) I/O resource that is described in terms of the functions it must be capable of performing, which is mapped by the system to a particular real resource at run-time. Such an approach implies that all available input devices, regardless of type (or location) are in some sense acceptable equivalents and all output devices are correspondingly equivalent.

The user must, when he so choosea, be able to decide what I/O devices of the ones available to him he wants used. In other words the user must be able to specify which phyaical resources the pseudo resourcea correspond to. It may also be necessary for the user to provide detailed I/O coding for the control of a device if such a device is not already known to the system.

The particular design approach taken in Multics is based on two practical requirements, one having to do with the system's responsibility for dispensing and recovery of all real I/O devices, and the other having to do with the run-time mapping of valid user-coded I/O operations, regardless of their degree of specificity, onto specific devices and in the manner and with controls appropriate to those specific devices.

First, it is recognized that at any given time, as a consequence of the I/O device needs of a process, certain specific I/O devicea (or device capabilities) must be exclusively allocated to apecific processes or sets of processes. The question of how the I/O System decides how to allocate devices, how to reclaim devices, and how to insure exclusive use of a device by the intended processes is largely independent of the central theme of this discussion, the structure of the I/O System, and, although important, will not be discussed here.

Second, any programmed I/O operation should at source level, at least, be expressed (coded) in a general way that specifies the I/O source or sink, not by its device designation but only by a placeholder name for that source or sink. (Moreover, as an added convenience to users, it may be possible to code certain standard I/O operations so that even this name may be inferred from context.)

For example, [and here we illustrate only schematically], rather than use a specific device designation such as in the following form:

```
read from "card_reader_2" into area_23;          (1)
or
read ("device 35", area_23);
```

we might instead say:

```
read from the stream named "Billy" into area 23;  (2)
or
read ("my_console", area_23);
```

depending on the syntax of the coding language being used.

Here in example (2), "Billy" and "my_console" are simply identifiers for sources of data. For such a read statement to have any meaningful effect, the specific device represented by that identifier must be bound to or "attached" to (i.e., associated in some way with) "Billy" or "my_console" at some time after the device is allocated to the process and before the read statement is executed. The Multics I/O System is responsible for maintenance and supervision of these device-source name associations. Similarly for output, names for sinks are used in write statements rather than actual output device designations. Thus by analogy to the read examples in (2) above we could conceivably picture something like

```
write ("his_console", "format 12", area_22);      (3)
```

in which "his_console" is here intended to suggest the name of some sink (output device). The attachment at any given time may be to one of a set of several (different) devices. Thus, if a single process had several consoles allocated, the process could simulate a "party_line" conversation on the several consoles where the name "his_console" could be attached and reattached, possibly cyclically, among the several different allocated devices.

The name chosen for elements of the set {source, sink} is stream. Conceptually, the attaching of a stream name to a particular device is a form of parameter binding. The device designation playa the role of the actual argument and the stream name that of the formal parameter. In order to apply more than one "argument" to the same "parameter" Multics provides for the detaching of a device (designation) from a stream name so that subsequently another device can be attached to the same stream name.

To carry out a read or write operation (call) of the type suggested in (2) and (3) above, the following steps can now be visualized. The system module that received and is responsible for "interpreting" this call must first perform a table look-up to determine the device designation (and type of device, constraint rules, if any, for use, etc.) that is currently associated with the named I/O stream parameter. In principle, assuming the I/O call parameters are consistent with the data kept in this so-called Attach Table, this same I/O control module can then convert this request into an I/O action -- i.e., by initiating the desired I/O operations after generating the required channel commands, etc. Because the system must be capable of supporting an open-ended number of devices, device types, and controllers, considerably more modularity is called for. So, in actual fact, the I/O control module (called the I/O switch) merely transmits the now more specific I/O request as a call to an appropriate "specialist" module, a Device Interface Module (DIM), for each type of device. A list of DIMs currently in general uae in Multics is given in Appendix B. This DIM in turn takes charge of getting the I/O request accomplished as auggested in Figure 2.

It is, therefore, the function of the DIM to convert the I/O request into a set of specific channel commands for the particular device associated with this DIM. The DIM knows both the conventions of the I/O System and the conventions of a particular I/O device and functions as a translator from one set of conventions to the other. In order that all devices may be fully exploited it is necessary that the I/O System "language" be carefully chosen. The I/O System calls of Multics are described more fully later and in Appendix A.

Description of the I/O System

The Device Interface Module converts a generalized I/O request into specific instructions understandable by a particular device. In doing this, it must compile a program for the hardware General Input Output Controller (GIOC) [6] which it can in turn supply to the target channel. The compiled program reflects the idiosyncracies of the particular device to which the stream is attached. It (the program) may include line controls in the case of remote terminals, select instructions in the case of tapes, and so forth. In addition, the DIM may need to convert the internal character code used by the system into an appropriate character code for the device. Typewriter terminals for example, come in many different varieties. Virtually every different variety has different character codes.

The Device Interface Module after compiling a program for the GIOC, calls a module that serves as an interface for the GIOC to start the I/O using this GIOC program. It is the DIM's responsibility to interact with the GIOC Interface Module (abbreviated as GIM) until this I/O request has been completed.

The GIOC Interface Module is responsible for the overall management of the GIOC. Thus, the GIM is also responsible for overall monitoring of the operation of the GIOC. This requires answering interrupts (i.e., that its code acts as an interrupt handler for), recognizing completion of tasks, and transmitting to its caller status information deposited by the GIOC.

It may be necessary for the DIM to wait for a particular I/O operation to complete and/or be awakened when it does occur. For this purpose an entry is provided in the traffic controller that causes the process to be suspended until it is reawakened. When the awaited operation completes, the GIM (which is invoked by a hardware interrupt from the GIOC) calls the traffic controller to awaken the suspended process. This is the interface between the traffic controller and the I/O System. All multiplexing of processors is, therefore, accomplished by the traffic controller.

The I/O System is implemented by a set of subroutine calla, twenty at present. The stream-DIM association is established by the attach call:

```
call attach (stream_name, DIM_name, device_name);
```

This call creates an entry in the Attach Table for the stream identified by stream_name, if one does not already exist, and associates the DIM identified by DIM_name with it. The DIM itself is then invoked to initialize (establish communication with the device and prepare it for further transactions) the device identified by device_name.

Once the device has been attached it may be utilized by issuing a read or write call:

```
call read (stream_name, buffer);
call write (stream_name, buffer);
```

Where stream_name identifies the stream with which the desired DIM and device are associated, and buffer indicates the area from which data is to be written or into which data is to be read. The I/O switch, upon receiving a read or write call, finds the entry in the Attach Table associated with this stream and invokes the associated DIM at the read or write entry. The read and write calls represent the primary means by which all data enters or leaves the system.

In order to dissolve an attachment the detach cal is used.

```
call detach (stream_name);
```

This call causes the association of the specified stream with any DIMs and devices to be dissolved. The I/O switch invokes the associated DIM which in turn terminates (releases the device and ends communication with it) the associated device or devices. When the DIM returns control to the I/O switch the stream-DIM association in the Attach Table is deleted.

There are many other I/O System calls which concern aspects of the I/O System that are not of immediate concern to this discussion. These include calls to set device modes (readable only, writeable only, forward spaceable only, etc.), calls to operate devices synchronously or asynchronously (e.g., readahead and writebehind), calls to establish input delimiters, calls to determine the current device status, and calls to reposition the current read or write position of a device (e.g., tape spacing). A short description of these calls is given in Appendix A.

A final I/O System call that is of interest here is the order call. This call provides the escape mechanism when an operation not implementable by any of the other generalized I/O System calls must be performed.

```
call order (stream_name, request_name,
            other_information);
```

This call is transmitted by the I/O switch to the appropriate DIM which performs the operation indicated by request_name making use of data supplied in other_information if necessary. Examples of order requests might be to repunch a card on a card punch or lock the keyboard of a console.

Up to this point discussion of input-output has been in terms of communication with physical devices. It has been shown that the only software that deals specifically with any single device is the DIM associated with that type of device. The I/O System, other than the DIMs, knows nothing of devices. It, therefore, follows that the I/O System does not necessarily have to communicate with a physical device but that DIMs may be written to operate on the data to be input or output in any manner whatsoever. Such DIMs are said to be associated with a virtual or pseudo-device and are termed pseudo-DIMS.

The most important pseudo-DIM is the File System Interface Module (FSIM) which treats a segment in the Multics File System as an I/O device. When a segment in the file system is attached to a stream via the FSIM, read and write calls on that stream will cause data to be read from or written into the segment. The FSIM provides the interface between the I/O System and the File System in Multics. However, unlike many systems this interface is not heavily used because the File System is usually called directly.

Another class of DIM is one that translates one I/O call to another I/O call, i.e., its pseudo-device is a stream. A stream that is used as a pseudo-device is termed an object stream. The most important of this class of DIMs is the "synonym" module. When an attachment is made via the synonym module the specified device is another stream. Any subsequent calls to the first stream is transformed by the synonym module to the same call on the latter stream. The stream names are, therefore, synonymous.

Applications

In the Multics system certain stream names are established, by convention, for normal use. The first of these is "user_i/o". This stream is normally associated with the user's primary I/O device, e.g., in a normal console session "user_i/o" will be attached to the user's console. Two other stream names are also established: "user_input" and "user_output". These streams are normally attached to "user_i/o" via the "synonym" module as illustrated in Figure 3a, i.e., they are made equivalent to "user_i/o". Since at present most programs that perform I/O intended to do so with the user's console, the stream names "user_output" and "user_input" are the ones used in calls to the I/O System in these programs. This illustrates one of the important purposes of the "synonym" DIM, to permit the manipulation of stream attachments without having to attach and detach physical devices. The streams "user_input" and "user_output" could normally be attached directly to the user's console as shown in Figure 3b. However, this would force the console to be detached whenever these streams were attached to some other device. Detachment and subsequent reattachment implies that certain physical hardware action has been taken with regard to the device. In the use of a console this might include termination of communication with the console and subsequently having to reestablish this communication. It would not be difficult to indicate to the DIM to keep the device active, however, the use of synonyms is more straightforward and makes more visible the states of various devices, i.e., if they are attached they are active. In other words, synonyms are an easy, efficient method of changing the binding of streams to devices. Because of this use of synonyms the "synonym" DIM has been highly optimized for the simple switching described above.

Some important and heavily used features of Multics serve to illustrate some of the advantages of this organization of the I/O System. A user of Multics may sometimes desire to redirect the output that could normally appear on his console to some other device. This situation usually arises because the output is lengthy and would require excessive amounts of time to print on a console. The Multics system provides a service by which the contents of segments in the file system may be printed on a high speed printer. Therefore, it is a fairly common occurrence for a user to redirect his output to a segment in the file system using the FSIM mentioned above so that it may be printed by the high speed printer or examined using a text editor. To do this the following I/O System calls must be made:

```
call attach ("file_output_stream", "fsim",
             "segment_name");
call detach ("user_output");
call attach ("user_output", "synonym",
             "file_output_stream");
```

The first call causes the segment, "segment_name", to become the receiver of all subsequent data directed to the stream "file_output_stream" by a write call. The second and third calls cause the stream "user_output", the stream on which all standard write calls are made, to be disassociated from "user_i/o", the stream associated with the user's console, and instead be attached to the new stream "file_output_stream". Again the use of synonyms is not mandatory but is included for the reason mentioned earlier. All subsequent output that would normally have appeared on the user's console would now be placed in the segment "segment_name". This new situation is depicted by the graph in Figure 3c.

There are many instances in which a user wishes to issue the same set of commands (a command is a line typed at a user's console requesting some action to be performed by the computer) many times. Rather than doing so manually he may instead put the set of commands in a segment and then cause this segment to be read as input one command at a time. This may be done by the following I/O calls:

```
call attach ("file_input_stream", "fsim",
             "input_segment_name");
call detach ("user_input");
call attach ("user_input", "synonym",
             "file_input_stream");
```

The segment whose name is "input_segment_name" contains the commands to be executed. The action performed by these calls is analogous to those performed by the above calls concerning output. All subsequent standard read calls will cause input to be taken from the segment "input_segment_name".

Consider now the situation that results when both the standard input and output streams are attached to segments simultaneously. In this case direct communication with the user has been eliminated. The user controls his process only indirectly through the input segment. A process that is in this state, i.e., whose standard input and output streams are attached to segments rather than to an interactive console, for its entire lifetime is called an absentee process (see Figure 3d). Absentee processes are the means by which background or batch jobs are implemented in Multics. The advantage of an absentee process from the system view is a better allocation of resources since absentee jobs may be scheduled at periods of low interactive demand. The point of interest here is that an absentee process, as opposed to an interactive process, is obtained by a few slightly different calls to the I/O System during process initialization and that no other special user or system programming is necessary.

In order to restore the situation to the interactive state just two I/O calls are necessary for each of the standard input and output streams. Thus for the input stream there would be:

```
call detach ("user_input");
call attach ("user_input", "synonym", "user_i/o");
```

Upon completion of these two calls the standard input stream is again attached to the user's console. The stream "file_input_stream" remains attached to the input segment.

The "synonym" DIM, as mentioned earlier, is one example of a DIM that uses another stream as the device upon which it acts. Such modules are effectively spliced into the flow of control in that each such module gains control and in turn passes control onto another DIM invoked as a consequence of its call to the I/O System on its object stream. The "synonym" simply results in an identical call to the object stream. However, such a DIM could easily perform some useful operation before passing the call on. A good example of such an operation is code conversion on the data to be read or written. A simple example could be to reformat a string of characters meant to be written on a console with a wide carriage for writing on a narrow carriage by properly placing carriage returns in the data.

Similarly such an intermediary could be used to make one device appear as another device. For example, if a light pen were to be added to the system as a new input device, a DIM could be written to make data read from a segment via the FSIM simulate the input from the light pen in order that all the associated software may be checked out before the actual installation of the device.

A final example of such intermediate modules is the broadcaster. This DIM allows fan out of I/O System calls. Rather than having one stream as its object, the broadcaster may have several. A call on a stream attached via the broadcaster is transmitted to all streams attached to this stream via the broadcaster. This is simply an extension of the synonym module. For example, a user may wish to record all the output typed on his console in a segment of the file system. To do this he simply attaches the stream "user_output" to both "user_i/o" and "file_output_stream" as indicated in Figure 3e.

Conclusion

It is the purpose of the Multics I/O System to permit I/O operations to be specified in a device independent manner, thereby permitting easy interchange of devices while programs are in execution. The designers of the I/O System have been able to achieve this goal largely because certain functions associated with I/O (file system, processor multiplexing) have been provided as independent facilities in Multics which are invoked by the I/O System as well as other programs. The method used to attain device independence is to define a set of I/O calls which are used to specify all I/O operations in a general manner. All devices are addressed symbolically by stream name and the binding of streams to devices can be modified dynamically.

The modular structure of the I/O System facilitates introduction of new devices. In order to logically add a device to the system, a user or system programmer need only provide the detailed I/O coding for that device in the form of a Device Interface Module. This ability to add new devices is necessary to assure the system's longevity.

Users of the I/O System, may if they desire, bypass the general mechanism. Instead of making a general I/O call, programs can invoke Device Interface Modules or even the GIOC Interface Module directly. The user who takes this approach loses the switching capabilities, device independence, and other advantages that the general mechanism provides. So far, no Multics user has needed or chosen to bypass the general mechanism. Some users, however, write their own DIMs making use of the order call to specify special requests.

The applications described earlier indicate some of the most common uses of the I/O System. The facilities of file input and output and absentee are achieved easily both conceptually and in practice and could not have been provided, in such a general manner, without device independence and stream switching. The I/O System has also proved very useful for system development, e.g., when testing a program that normally uses the high-speed printer it is advantageous to use a less critical more accessible device than one of the two printers available. The capabilities present in the Multics I/O System, as described here, have, therefore, proved well worth the careful design effort necessary for its development.

Acknowledgement

During the many years since the Multics project began a great number of people have contributed in the formulating of ideas for the I/O System. People who have contributed significantly to this effort are F. J. Corbató, R. C. Daley, S. I. Feldman, E. L. Glaser, D. Levenson, J. Ossanna, D. Ritchie, J. H. Saltzer, and V. L. Vyssotsky. The authors would also like to acknowledge the work of S. Dunten, N. I. Morris, T. Skinner and D. Widrig for their work in designing the GIOC Interface Module.

Appendix A

The following is a list of general I/O System calls and a brief description of their functions. This list serves only as an indication of the type of operations that are thought to be necessary in Multics, not as a complete description of their operations. Complete descriptions are given in [7].

attach establishes an association between a stream name, a device's control software (DIM), and a device. All subsequent operations on this stream will invoke the associated control software and will be performed on the associated device.

detach destroys an association created by an attach call.

read causes input to be taken from the device associated with the given stream and placed in the indicated buffer area.

write causes output to be taken from the indicated buffer area and written to the device associated with the given stream.

seek modifies the current position of the read and write pointers for the device associated with the given stream.

tell returns the current position of the read and write pointers for the device associated with the given stream.

changemode changes the current mode of the device associated with the given stream and returns the old mode. Modes determine attributes of a device such as whether reading or writing is permitted.

readsync determines whether or not the DIM associated with the given stream will perform read-ahead on the associated device. Performing read-ahead is to read input from a device before the read call is issued.

writesync determines whether or not the DIM associated with the given stream will perform write-behind on the associated device. Performing write-behind is to write output on a device after the write call has returned.

resetread erases all currently accumulated read-ahead from the device associated with the given stream.

resetwrite erases all currently accumulated write-behind intended for the device associated with the given stream.

worksync determines whether the device associated with the given stream is in workspace synchronous or asynchronous mode. Being in workspace synchronous mode means that when a read or write call returns, the I/O System is finished using the provided buffer area associated with this call. If the call was a read call the desired input has been placed in the buffer area. If the call was a write call the data has been taken from the buffer area. Being in workspace asynchronous mode means that buffers may still be in use by the I/O System after the call has returned. If a read call then the buffer area may not yet contain the desired input, but it will be filled in at some later time. If a write call then the data may not yet have been taken from the buffer, but the I/O System will do so at some later time. Workspace asynchronous mode allows programmers to perform asynchronous I/O transactions and multiplex their I/O calls.

upstate returns the current status of a specific asynchronous transaction on the device associated with the given stream.

iowait returns the current status of a specific asynchronous transaction on the device associated with the given stream. The iowait call will not return until the indicated transaction is complete, i.e., the I/O System is finished with the buffer area.

abort · causes the indicated transaction or transactions on the device associated with the given stream to be aborted.

getdelim returns the current break characters and read delimiters for the device associated with the given stream. Break characters define the extent of canonicalization and erase and kill processing of input [7]. Read delimiters determine on which input characters a single read call is to cease reading.

setdelim modifies the current break characters and read delimiters for the device associated with the given stream.

getsize returns the length, in number of bits, of the size of a basic element to be read or written on the device associated with the given stream. For example, Multics uses seven bit ascii right adjusted in a nine bit field as its standard character set so the element size for character oriented devices is 9.

setsize modifies the element size for the device associated with the given stream.

When a specific function on a specific device cannot be logically specified by any of the above general calls the order call is used:

order is used to specify device dependent requests to be executed by the DIM associated with the given stream. Examples include locking the keyboard of a console and unloading a magnetic tape.

Appendix B

The following list briefly describes the Device Interface Modules (DIMs) generally available and widely used in Multics. Detailed descriptions are given in [7].

Typewriter DIM - currently operates all devices used as user consoles in Multics. These include Teletype Models 33, 35, and 37, IBM 1050 and 2741, Datel 30, ARDS, and Terminet 300.

Synonym DIM - causes two streams to become synonymous, i.e., all I/O calls (except attach and detach) on either stream result in the same I/O operations being performed.

File System Interface Module - causes segments of the file system to be treated as input and output devices.

Multics Standard Tape DIM - is used for reading and writing tapes in Multics standard tape format.

Nonstandard Tape DIM - is used for reading and writing tapes in any format.

Card DIM - is used for reading and punching punched cards.

Printer DIM - is used for writing to the high speed printers.

ARPA Network DIM - is used to input and output from the ARPA Network of which the M.I.T. Multics installation is a part.

Communications Line DIM - is used to read from and write to a dedicated PDP-8 over a high speed communications line that is connected to the M.I.T. Multics installation. This PDP-8 is used for monitoring of Multics and for graphics.

References

[1] Daley, R.C. and Neumann, P.G., "A General-Purpose File System for Secondary Storage", AFIPS, 1965 Fall Joint Computer Conference, Vol. 27, Part 1, Spartan Books, Washington, D.C., pp. 213-229.

[2] Saltzer, J.H., "Traffic Control in a Multiplexed Computer System", Sc.D. Thesis, Department of Electrical Engineering, M.I.T., June (Available as M.I.T., Project MAC Technical Report No. 30).

[3] Lett, A. and Konigsford, W., "TSS/360: A Time-Shared Operating System", AFIPS, 1968 Fall Joint Computer Conference, Vol. 33, Part 1, MDI Publications, Wayne, Pennsylvania, pp. 15-28.

[4] CP-67/CMS User's Guide, IBM, October, 1970.

[5] System/360 Operating System Concepts and Facilities, IBM, Form 128-6535-1, June, 1967.

[6] Ossanna, J.F., Mikus, L., and Dunten, S., "Communications and Input-Output Switching in a Multiplex Computing System", AFIPS, 1965 Fall Joint Computer Conference, Vol. 27, Part 1, Spartan Books, Washington, D.C., pp. 231-242.

[7] Multics Programmers' Manual, Preliminary Edition, M.I.T., April, 1971.

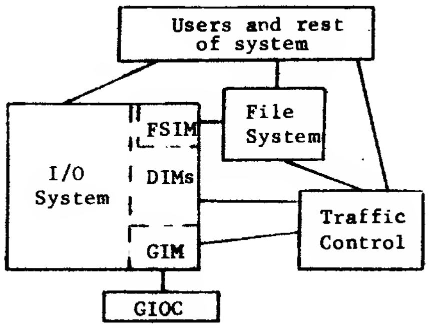


Figure 1 - The I/O System's relationship to some other important Multics facilities.

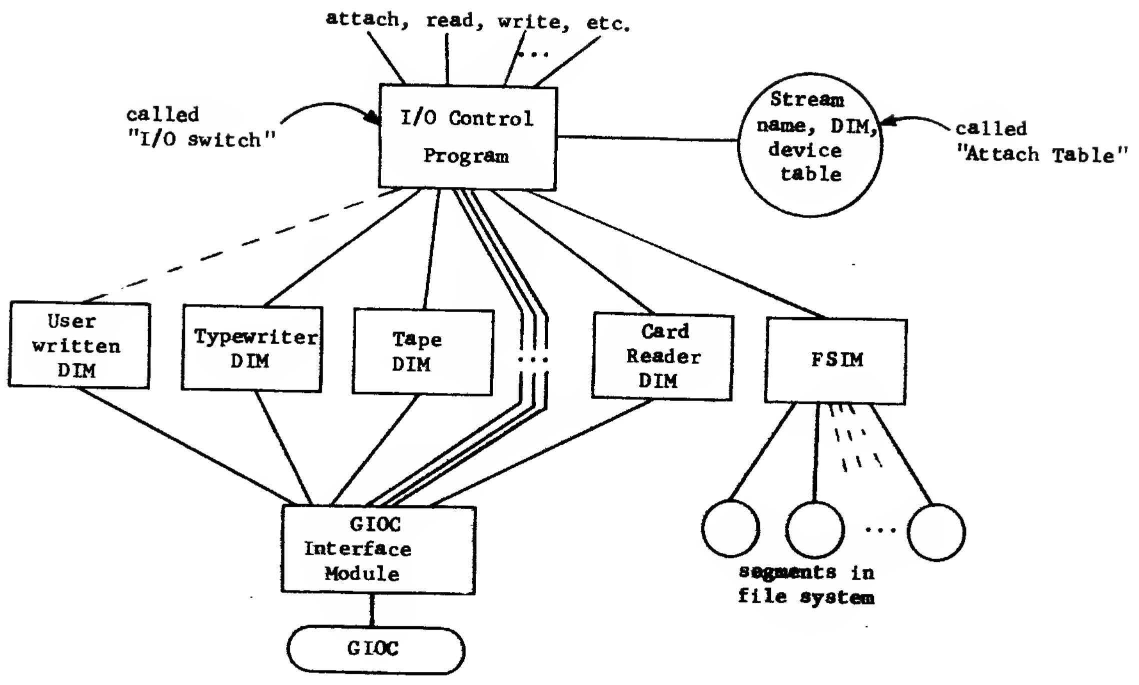


Figure 2 - Simplified view of I/O System organization.

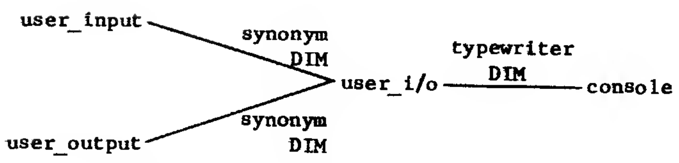


Figure 3a - The standard attachment graph.

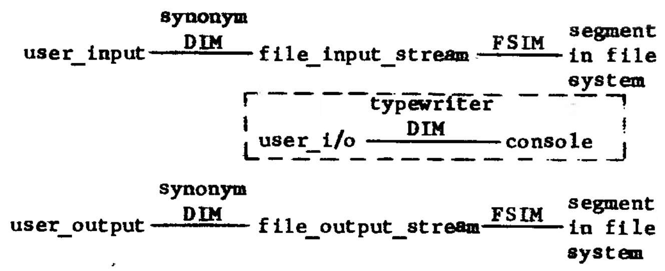


Figure 3d - Absentee attachment graph. For a true absentee process that has never been attached to a console the attachment in the dashed box is unnecessary.

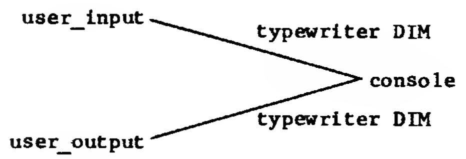


Figure 3b - A standard attachment graph without the use of the synonym DIM.

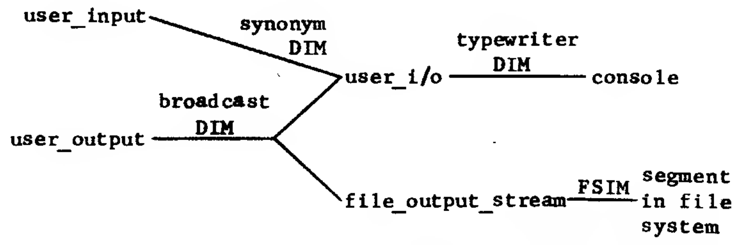


Figure 3e - Attachment graph with standard output written to both the user's console and a segment in the file system.

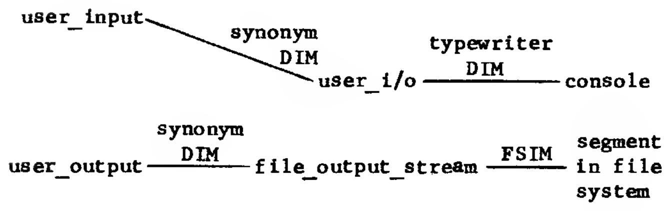


Figure 3c - Output attached to a segment in the file system.

CHAPTER 3

BEGINNER'S GUIDE TO THE USE OF MULTICS

September 20, 1973

There are a large number of ways to use Multics. You may, at different times, find that you are using a program preparation facility, or a program debugging facility, or a memorandum typing facility, or a management information facility. One of the interesting properties of Multics is the ability for a knowledgeable programmer to construct a single program which makes use of several of these facilities at once. For the beginner, however, the problem is simply to figure out which one of several ways of doing something is appropriate for his project. In this chapter will be found a guide to typical ways of using Multics and its most commonly used facilities, and a number of examples of sessions at the terminal, to give a feel for the way one fits things together to achieve useful results. We must begin by exploring a number of issues having to do with the simple mechanics of using the system.

The Mechanics of Terminal Usage

Although there are several different varieties of typewriter or graphic terminals which can be used with Multics, they all are used in similar ways; the way in which Multics normally expects these terminals to be used is our subject here. Note that Multics permits a subsystem designer flexibility to change conventions which are not exactly suited to his needs. Therefore, we will describe here the standard conventions which apply to ordinary use of Multics, and which are also used by most programs. Indeed, an important property of Multics is the extent to which the mechanics described here are universally used by so many different parts of the system.

Most computer terminals are designed with flexibility to allow use with different kinds of systems. This flexibility is expressed in the form of switches whose setting must be correct if proper operation is expected. For example, the IBM model 2741 terminal may have one or two switches on the left side, one labeled "com-lcl" (which must be set to com), and the other labeled "inhibit auto-eot" (which, if there, should be set to on). For switch settings on other terminal types, see the MPM Reference Guide section, Protocol for Logging In.

The connection of the terminal to the computer is accomplished by ordinary telephone lines, and by dialing the telephone number of the computer. This number is usually equipped to automatically connect you to the first free line into the computer system. Multics is designed to inform potential users that it is fully loaded by printing a message on the terminal rather than by refusing to answer the telephone or returning a telephone busy signal. Either of these latter two responses to dialing Multics is a symptom of trouble and should be reported.

Communication of keyboard characters with the computer is accomplished by conversion of these characters into sequences of tones which can be sent over the telephone line. The piece of hardware which does this conversion is called a _dataset_ or _modem_ (for _mo_dulator-_dem_odulator); there must be one modem at the terminal and another at the computer. Two types of modems are frequently found associated with computer terminals: those directly attached to the telephone line, and those which are acoustically coupled by inserting the telephone handset physically into the modem. The directly attached devices normally come with a special telephone set which has a row of buttons; one of these buttons must be depressed in order to get a dial tone to start the call. In contrast, the acoustic coupler is designed to work with any ordinary telephone anywhere.

After dialing the Multics telephone number, you should hear one or two rings, and then the computer will answer. The next step is to complete an electronic handshake sequence, first between your modem and the one at the computer, and then between your computer terminal and the Multics terminal controller. The computer starts the sequence immediately after it answers by placing a tone which you can hear on the telephone line. You should then press the _data_ button on the modem, if the modem is directly attached, or else insert the telephone handset into the acoustic coupler. The handshake sequence should then proceed to completion all by itself, with a characteristic pattern of clicks and gurgles that you will soon learn to recognize as normal operation, ending with a printed message from the computer.

There are several possible ways in which the handshake sequence may fail. Before giving up, check the following list of possibilities:

1. Are you sure the computer answered and provided the initial tone? If not, check to see if Multics is in operation.

2. Is the terminal plugged in and is its power switch _on_?

3. Is the cable connecting the terminal to the modem properly in place?

4. Is the modem plugged in? (If it is an acoustic coupler, it may have to be turned _on_ also.)

5. Are all the switches on the terminal and modem in correct position?

6. Did you dial the correct telephone number? Generally there are different numbers for different terminal types and speeds. Check your telephone number list.

7. Has the terminal in question ever been used with Multics before? If not, possibly it is missing some feature required for use with Multics.

8. Has this telephone line ever been used with this acoustic coupler before? Possibly the line is too noisy or weak for the brand of coupler used, or maybe there is too much amplification in the telephone line and one of the modems is being overloaded.

9. Try hanging up and dialing again once or twice. With the array of equipment between you and the computer, flukes are common.

If all of these checks fail to turn up anything, it is time to turn to expert help.

Assuming that the handshake was successful, it was completed with the printing of some message from the computer, e.g., "Multics version 15.11". You are now in communication with the computer, and anything you type on the keyboard will be both printed and heard by the computer. Whenever Multics or any program prints anything to you, the keyboard will be temporarily locked, thus preventing you from typing anything. At all other times, the keyboard is unlocked, and you are free to type.

Generally, you will type messages with the intent that they be read and understood by some program; you should always keep in mind just exactly which program will be interpreting each message you type.* To start with, the system has arranged that your input lines will be directed to a login program which will insist that you type information properly identifying yourself. The login program will, at one point, exercise a special feature of your terminal by disconnecting your keyboard from your printer, so that you may type a password without producing a printed copy. (If your terminal doesn't have this feature, the login program

---

* It is important to realize that you are allowed to type even if some previously initiated operation has not finished yet and technically the system or subsystem is not ready for another typed line from you. If you can anticipate your next input lines, you may type them at any time; they will be stacked up and used, in order, to satisfy future requests for input from you. This feature permits you to work ahead of the computer, and overlap your thinking and typing with waits for response from it.

will instead print some random letters on the paper in the place you are to type your password.) When the login program is satisfied that it knows your identity, it will start a program known as the _listener_ which is usually used to supervise your entire terminal session. The listener interprets lines that you type as names of programs you wish to run. Whenever the listener is listening for input, the terminal is said to be at _command level_. The programs which you ask the listener to call are known as _commands_. Command level is an important reference point, and we will use this term frequently. Several of these ideas may come into better focus in the example terminal sessions which appear later in this chapter.

It is common, as well as human, to make typing mistakes, so two correction conventions are normally in operation at all times. One of them allows you to erase, so to speak, and then retype small typing mistakes, and the other allows you to simply discard more extensive typing disasters. The erase convention uses the number sign (#) character. Whenever you realize that you have typed a character in error, type as the next character after it the number sign. When the line is read, it will be scanned for number signs; if one is found, it, and the character before it will be discarded; the resulting line is then assumed to be the line you intended to type. Two consecutive number signs will erase the two immediately preceding characters, and so on. Note that you do not correct errors by backing up and overtyping, as in some systems. If you do backspace and overtype, the system will presume that you want that particular combination of overstruck characters to be in your input line. In this connection, note that the system is more concerned with the appearance of the final printed line on your terminal than it is with the order you typed things in. Thus, for example, the order in which you produce overstruck characters is unimportant, and extra up and down case shifts are ignored.

If you notice a serious error farther back in the line you are typing, you could correct it by typing enough number signs to erase everything back to and including the error, and then retyping everything that was erased, this time correctly. However, it may be simpler to just type a _kill_ character (the commercial at sign, @). When this character is encountered in an input line, it, and all of the line to the left of it are discarded. The corrected line is then retyped directly to the right of the kill character. Several examples of the use of erase and kill characters appear in the annotated terminal scripts later in this chapter.

Unless one is using a special program which has arranged things differently, the unit of communication with the computer for the typist is the completed line, ending with the function key which returns the carriage to a new line. Thus, typing "new line" is the signal that the typist is satisfied with the line as it stands; the line is scanned for erase and kill characters, and then passed along to satisfy the next request for input.

Some terminals do not have all of the 96 different characters which can be typed in to Multics programs. For example, the IBM 2741 terminal does not have square brackets. There is a set of conventions which allows one to type something else which means the same thing. All of these conventions use one special character as an _escape_ character to indicate that the next character is to be interpreted differently than usual. On a 2741, the escape character is the cent sign (¢). If one types a cent sign followed by a "less than" sign, these two characters together will be taken to mean a left square bracket. A complete set of escape conventions which apply to your terminal may be found in the MPM Reference Guide section, Typing Conventions.

Finally, two emergency measures should be mentioned. Every terminal has somewhere on it a special button which is always pressable, even if something is being printed and the rest of the keyboard is locked. This button is called the _quit_ button, and, when pressed, will cause the system to stop whatever program was running and return to command level. In this way, even if you have started a runaway or incorrect program, you may always keep positive control of the situation. Note that when the quit button is used, the work in progress, while halted, will not necessarily be saved if you begin doing something else. Generally, unless you take special measures, you will find that pressing the quit button discards all work which was done since the previous time you were at command level.

The second emergency measure is the terminal disconnect. If you should happen to turn off the terminal power, or hang up the telephone while logged in, the system will first perform the equivalent of a quit, then it will automatically perform a logout command. Of course, it can not print the usual logout message on your disconnected terminal. In general, you need not worry about disrupting the system by such an abrupt disconnection, but your own work may be lost back to the last time you were at command level.

A Multics Terminal Session

Having accumulated some familiarity with the basic mechanics of using Multics, the easiest way to proceed to familiarity with the system itself is to look over the shoulder of an experienced but cooperative user, and pester him with questions about what seems to be happening. The closest alternative we can achieve here is to walk through some sample terminal sessions, explaining in some detail the various pieces of an emerging picture. To start with, we will consider one of the simplest possible sessions, in which a user logs in to the system, checks on the latest news and notices, uses the system as a desk calculator to balance his checkbook, and then logs out. Later examples will illustrate typing and editing information and use of the Multics storage system. To begin with, however, the simple terminal session illustrated in Figure 3-1 will allow us to decouple from those considerations the purely mechanical issues underlying all

```
1       #
2
3       Multics 15.8; MIT, Cambridge Mass.
4       Load = 55.0 out of 60.0 units; Users = 58
5*      login Williams
6       Password:
7*
8       Williams Apollo logged in:  09/29/70  2139.4 edt Tue
9       Last login 9/28/70 1633.2 edt from terminal "209"
10      New or updated help segments:  pl/1_status, tty_bug, news
11      r 2139  3.914  12.070  231
12
13*     help nes#ws
14      (10 lines follow)
15      09/29/70
16      The following changes were made in the on-line system today:
17      1)  The editor command, edm, was replaced with a new version
18          which eliminates a bug encountered when input lines
19          overflow its input buffer.
20      2)  A new command named change_default_wdir (abbreviated cdwd)
21          was installed.  This command changes the user's default
22          working directory for the duration of the current
23          process or until the command is issued again.
24      (end)
25
26(*)   more help?  yes
27      (68 lines follow)
28      Following is a summary of all system changes made 9/1 to 9/28:
29      9/28   Replaced PL/I compiler, removing varying string bug.
30      9/26  Added 12 million words of disk stora
31      QUIT
32      r 2142  1.667  4.760  110
33
34*     decam
35      Go
36*     =0
37*     +14791
38*     +38525
39*     -2741
40*     -3482
41*     -49768
42*     p
43      -2675
44
45*     q
46      r 2148  .515  4.040  135
47
48*     logout
49
50
51      Williams Apollo logged out 9/29/70 2149.1 edt Tue
52      CPU usage 5 sec
53      hangup
54       -
```

Figure 3-1: A Sample Terminal Session.

use of the system. In each of our examples, we will take replicas of actual terminal sessions, and add line numbers down the left side so that we may refer to them. We have placed an asterisk (*) beside those lines typed by the user; the remaining lines are those printed by whatever program he is communicating with. The session starts at an IBM 2741 terminal, immediately following the dialing of the Multics telephone number.

The login sequence, all by itself, raises a fairly large collection of issues. Let us examine this script, line by line. Line 1 was printed as a consequence of the electronic handshake sequence between the typewriter control program and the 2741. In order to establish what kind of terminal has called, the control program tries several experiments, attempting to elicit a response from the terminal. One of the experiments caused the terminal to print a number sign. That experiment being successful, the terminal type was identified, and the system printed a greeting message on lines 3 and 4, after putting in a blank line (line 2) to insure that the carriage is at the left edge and that anything accidentally printed by the experiment is separated from the message. Note that a line from the computer usually ends with a "new line", so that the next message, whether typed by the user or the computer, starts at the left edge of a new line. The second line of the greeting message (line 4) tells the number of users currently logged in, and the load they are placing on the system. The average user places a load of 1.0 load units on the system, and in this example the hardware configuration in use will support 60 units, or 60 average users. Some users with restricted command repertoires may be rated at less than 1.0 load units; others may be rated higher. Since the load, 55.0, is well below the limit, 60.0, we will have no trouble logging in. If the load were equal to the limit, we might still attempt to log in; it may be that some part of the load can be deferred or some low priority user could be asked to stop working. After printing line 4, the system unlocked the typewriter keyboard, and the user had two minutes in which to log in to the system. Thus, on line 5 he typed a login line, giving the personal name by which he is identified throughout the system. Note that the distinction between upper and lower case letters is significant in Multics input and output. If he had typed his name without the initial capital letter, it would not have been recognized.

Some users may type other things after their name. Such extra input items are necessary only if the user works on more than one project or charges his usage to more than one account, and then only if he does not want to use his standard billing or project identification for this terminal session.

On line 6, the login program responded by requesting the private password which is associated with the user's name. At this point, the program turned the terminal printing mechanism off and although our typist typed in his password on line 7, there is no printed record of it. Note that, as usual, he signaled that he had completed typing by typing a "new line", so the next message from the computer was printed on line 8.

Lines 8 through 12 are the response of the login program to the successful identification of the user. Line 8 records the date, time, name, and project affiliation of the user. The project affiliation refers to a grouping of users who are working together on a single project and therefore require frequent access to each other's information. As we shall see in the example terminal session exhibiting storage system usage, since the privacy system recognizes the existence of such groups, one can grant access to all members of a group by stating just the project name of the group. Line 9 tells the user of the conditions of his previous terminal session, so that he may discover if someone else is using his password.

Line 10 is called the <u>message of the day</u>. This message is updated frequently to reflect any important news for users. Rather than printing the details of the news here, though, the message usually refers the user to information files which may be printed with the help command. We will see examples of how to use this very handy facility in a moment.

Lines 11 and 12, the last lines printed as a result of logging in, are known as a <u>ready</u> message, since its appearance indicates that the terminal is now at command level, and that the command language interpreter is ready to start interpreting commands. The four numbers printed in the ready message have the following meanings:

2139	Time of day, in 24 hour form, to the nearest minute (e.g., 9:39 p.m.).
3.914	Number of seconds of central processor time used since the last visit to command level.
12.070	A measure of the memory used since the last visit to command level. It is intended to measure memory usage in a manner that is independent of system load.
231	Number of pages (1024 word blocks) of information brought in to primary memory since the last visit to command level.

A blank line, in this case on line 12, is printed as part of the ready message, to provide separation between successively typed commands. As we shall see, a ready message is printed every time that the terminal returns to command level. The information printed in the ready message, in addition to providing an occasional time stamp on one's terminal output, is frequently handy in estimating the relative cost of a just-completed operation, or in comparing the cost with another way of doing the same thing. (Note: for the uninterested, there is a special feature which can be used to suppress the ready message. For details, see the write-up of the ready_off command in the MPM Reference Guide Command section.)

At this point, the system has now created a process for the user. A process may be thought of as a private computer, working in its own memory, or address space, under control of the user at his terminal. The process has begun running in the listener program, so any line typed by the user will be interpreted as a command, that is, an instruction to call some program either belonging to the user or else in the Multics library. Our sample session continues as the user types his first command line.

The command line typed on line 13 illustrates three things: invoking a library program by name, passing that program an argument, and correction of a typing error. The user chose to follow up the suggestion given by the message of the day back on line 10, so he typed the name of the help command. That command is capable of giving help on a variety of topics; one selects the topic by giving the help command an argument, which names the desired topic.* The help command takes that argument as the name of a file of information which it then uses as a source of text. In this case, our user wanted to see the latest system news, so he tried to type the argument "news" following the command name "help". Unfortunately, he slipped up, and typed "nes". He then noticed his error, and typed the erase character (#) followed by the correct letters. Thus the line actually interpreted by the listener reads "help news".

The help command then replied by printing, on line 14, a notice of how much output was coming, and then on lines 15-24 the latest message from the on-line news file. After completing that message, it inserted a blank line (line 25) to improve readability, and then asked the user if he wished to see more. This question, on line 26, illustrates that some lines printed by the computer need not end with a "new line". After printing the question mark, the program printed two spaces, then stopped to await the reply of the typist. The parenthetical asterisk to the left of line 26 is intended to call attention to the fact that the typist only typed the last part of this line, namely the letters "yes", and the "new line".

Then, on line 27, the help program again printed a notice of how much output was coming, and proceeded with the next older set of news. Our user, not wishing to wait while 68 lines of information were printed, allowed the printing to proceed only until he saw news he had seen before, on line 30. In the middle of that line he pressed the quit button. The system responded immediately by printing a "new line", the word QUIT on line 31,

* If one does not even know enough to name a topic on which he needs help, typing "help" with no arguments will provide a tutorial on the on-line information currently available.

and a standard ready message on lines 32 and 33. The terminal was thus forcibly returned to command level, the help command having been suspended in mid-operation. The user was then ready to type his next command, on line 34.

There our user typed in the name of a desk calculator command program found in the Multics library. This command turns his terminal typewriter into a kind of simple adding machine, so that he can balance his checkbook. The desk calculator acknowledges that it is listening for input by printing the word "Go" on line 35. Our user, being experienced in the use of the calculator, proceeded to type in a whole series of requests to it on lines 36-41, first to clear its memory, then to add and subtract several numbers found in his checkbook. Note that he did not wait for a response to one request before typing the next one; he knew that the calculator does not reply to requests for memory clearing, addition, and subtraction. In fact, it is likely that he typed at least some of his input lines before the calculator was ready for them; he and the desk calculator were making effective use of the Multics type-ahead ability mentioned before. Finally, on line 42, he typed a request to print the result of all that addition and subtraction. This time, he waited for the response, which the desk calculator printed on line 43, followed by a blank line for readability on line 44.

Our user was then finished with the desk calculator, and wanted to type more commands; in order to return to command level, he typed the request q (short for "I quit") to the desk calculator on line 45. The calculator program responded by returning to its caller, and the terminal was returned to command level as the ready message on lines 46 and 47 attests.

Our user, having solved his immediate problem (there seems to be little Multics can do about the negative balance in his checkbook), then typed the logout command on line 48. The logout command, in addition to printing the messages on lines 49-53, took care of various housekeeping chores, such as updating accounting records and removing the user's name from the list of those currently logged in. It also triggered a telephone line disconnect sequence, which caused the minus sign to print on line 54. Note that although our user was logged in for almost ten minutes, he used only five seconds of the central processor's time. Such ratios are the basis for developing a time-sharing system which is to be used by a large number of people simultaneously.

With this example, we have now walked through an entire terminal session. If you wish, you might want to try to imitate this session the first time you log in, substituting your own name for that of our sample user. One thing that you would surely notice if you tried that experiment is that the ready messages would not be exactly the same as in our sample script. It is normal to observe a variation in the amount of processor time or number of page movements required to accomplish the same job several times. The variation arises because the system attempts, as often as it possibly can, to run your program on the coattails of other users, utilizing pages in common. To the extent that such sharing is successful, the charges to individual users may be reduced, but the size of this effect will vary with circumstances. Also, when the system is heavily loaded, it is harder to locate the resources required to run a program; the extra effort required shows up as a charge to the user who asked for them.

In addition to the commands illustrated here, you might try typing the help command with no arguments, and you might also try the who command. The Reference Guide gives complete information on many options and variations on these as well as on the commands illustrated in our sample scripts.

Typing and Editing Information

Probably the single most common activity of a user of a time-sharing system is typing in and editing information, with the intent that the information be stored for later use. One important property of a system which is normally approached by means of a remote terminal must be that it can store information from one usage session to the next. If this property were lacking, it would be unreasonable to use it to tackle any information processing job which could not be completed in a single sitting. Since that kind of restriction is unwanted, Multics provides an extensive system for storing and organizing information, the Multics storage system.

The unit of information which is stored, named, protected, and shared in the Multics storage system is known technically as a segment. One or more segments containing related information is usually called a file. Typically, a segment might contain a complete program written in the PL/I language, or a memorandum, or a collection of closely related data. We will return later to a variety of examples of how segments are named, protected, classified, and shared; for the moment we are merely interested in the mechanism by which one creates a brand new segment or modifies the contents of an old one. This mechanism is important because most subsystems which require substantial quantities of input expect to find their input in segments. For example, one uses the Multics PL/I compiler by first constructing a segment which contains the desired PL/I source program. Then he instructs the compiler to translate the source program found in that segment.

Segments which contain only strings of characters, and thus can be printed by a standard printing procedure without decoding their format, are known as printable segments; a PL/I source program is an example of a printable segment. All other segments may be categorized as binary segments, which is just a way of saying that they consist of a collection of bits which somehow represent information in a way different from the standard printable form. Usually, binary segments are created and read only by programs. Because they can be easily printed, printable segments are creatable, modifiable, and readable by human beings as well as by programs.

For the purposes of creating and editing printable segments, several general-purpose _editor_ commands are available. The two standard editors are named edm and qedx. The first, edm, is easy to learn and use, but limited in its repertoire of facilities when complex but methodical changes to a segment are needed. The second, qedx, is more powerful and is controlled by a concise input language, but is somewhat more difficult to master at first. Some subsystems (for example, BASIC and APL) provide their own built-in editor program in order to minimize the distinction between program creation and execution. We will here concentrate on the simpler of the two general-purpose editors.

As before, it is easiest to explain the operation of an editor by looking at a sample terminal session. In the example in Figure 3-2, edm is used to type in a new segment containing a short poem. We begin our reference line numbers from 1, realizing, of course, that the user who typed in this segment must have first logged in as in our earlier example. As before, we have marked with an asterisk lines typed by the user.

On line 1, our user typed the command to invoke the editor. Since the editor is willing to edit any text segment in the system, it is necessary to indicate which segment is to be edited. This indication is made by typing the name of the segment as an argument following the name of the editor command itself. In this case, our user has chosen the name, poem, as the name he would like to use for the segment he is about to create. On line 2 the editor replies with the observation that it did not find a segment named poem already in existence, so it assumed that it was supposed to create a new segment with that name.

To understand the message printed by the editor on line 3, we must realize that this editor operates in one of two modes: input mode, and edit mode. In the input mode, everything typed by the typist is presumed to be information to be stored in the segment. In edit mode, the typist's lines are instead taken to be requests to make changes to the already stored segment. Since the segment had not yet been typed in, the editor assumed we should start in input mode, which it signified by printing "Input." on line 3. As we shall see, when the editor detects that the typist is working on an old segment, it starts him off in edit mode instead.

Lines 4-8, then, are the intended information content of the segment, supplied by the typist. Note the use of an erase character near the beginning of line 5, to change the i to an o, and the kill character used on line 7 after noticing a blunder earlier in the line. Even though only one character was in error (the r should have been an e), it was necessary to type the

```
1*      edm poem
2       Segment not found.
3       Input.
4*      There was a young lady from Niger
5*      Who ri#ode with a smile of a tiger.
6*      They returned from the ride
7*      With thr lady@With the lady inside
8*      And the smile on the face of the tiger.
9*      .
10      Edit.
11*     t
12*     l smile
13      Who rode with a smile of a tiger.
14*     c /of/on/
15      Who rode with a smile on a tiger.
16*     t
17*     p 1000
18      No line.
19      There was a young lady from Niger
20      Who rode with a smile on a tiger.
21      They returned from the ride
22      With the lady inside
23      And the smile on the face of the tiger.
24      EOF
25*     w
26*     q
27      r 2024  1.280   5.284  225
28
29*     edm poem
30      Edit.
31*     l tiger
32      Who rode with a smile on a tiger.
33*     l
34      And the smile on the face of the tiger.
35*     i                     -- anonymous
36*     t
37*     .
38      input.
39*     A poem:
40*
41*     .
42      Edit.
43*     t
44*     p 1000
45      No line.
46      A poem:
47
48      There was a young lady from Niger
49      Who rode with a smile on a tiger.
50      They returned from the ride
```

Figure 3-2: An Example of Typing and Editing Information.

```
51     With the lady inside
52     And the smile on the face of the tiger.
53                              -- anonymous
54     EOF
55*    t
56*    c 1000 /tiger/giraffe/
57     Who rode with a smile on a giraffe.
58     And the smile on the face of the giraffe.
59     EOF
60*    t
61*    l anon
62                              -- anonymous
63*    d
64*    w
65*    q
66     r 2026  .875  2.132  150
67
68*    print poem
69
70                             poem    10/31/70  2026.7 est Sat
71
72
73     A poem:
74
75     There was a young lady from Niger
76     Who rode with a smile on a giraffe.
77     They returned from the ride
78     With the lady inside
79     And the smile on the face of the giraffe.
80
81
82     r 2025  .377  1.298  37
83
```

Figure 3-2 (continued)

entire line over again because, you may recall, the kill character deletes everything to its left on the same line.

Having completed the initial typing of the poem, our typist now wished to switch to edit mode. Now he was up against a slight problem: everything he typed was supposed to be stored in the segment. How was he to communicate to the editor program his intent to stop using the input mode? As we might expect, a trick is used. The editor checks each line typed in input mode. When it sees a line containing nothing but a period, it takes that line to mean that the mode should be changed, and it does not store that line in the segment being created. (Note that this means that one cannot store a line containing only a period while in the input mode. However, one can create such a line in edit mode.)

Thus, on line 9, we see only a typed period, and on line 10 we see the response of the editor, saying that "Edit." mode is now in operation. At this point, our typist, having looked over the printed copy of his input, noticed that he made an error on line 5--the word "of" should have been typed as "on". To make such changes easy to manage, the editor maintains a _pointer_, which is always pointing to some place in the stored segment. The typist may move this pointer from line to line, by issuing various requests. Thus, when our typist issued the request to switch to edit mode, the pointer was pointing to the last line he had typed. The t (for top--most edm requests are one letter mnemonics) request on line 11 moved the pointer to the top of the segment, ahead of the first line. The l (for locate) request, on line 12, started a search for the next line containing the string of letters "smile". When it found such a line, the editor printed it on line 13, and left the pointer pointing to that line. This operation of moving the pointer by searching for a string of letters is known as editing by _context_.

Having got the pointer set to the line which contained the error, our typist then issued a c (for change) request on line 14. The change request is designed to avoid the need for typing the whole line over, by mentioning first a string of characters which appears in the line, and then giving another string which is to replace the first one. What the typist wanted to express is the notion "change the string of letters 'of' to the string 'on'". Since, in general, one or both of the strings may contain blank spaces, we must invent some convention for communicating to the change request exactly what string is to be used for matching, and what string is to be used in the first string's place. The convention used is for the typist to choose any character he wishes that is not in either string -- his choice is called the _delimiter_ character. (The slash mark is often used since it is convenient to type.) Then he types that character three times, with the two strings in between. Thus, the substitution was expressed to edm by typing the request name c, followed by a space, then the first delimiter (/), the string of characters to be matched (of), then a second delimiter, then the new string to be substituted in place of the matching string (on), and finally a third delimiting character. In return for this input sequence, the editor performed the requested substitution, then printed the changed line to verify that the correct change occurred.

(Note that while editing by context is very convenient, context is often ambiguous, and one must constantly check to insure that the correct context was used. Thus, the word "of" might have appeared twice in the line; in that case, the change request would have changed both occurrences. If one wanted only the second occurrence changed, he would have to type a larger identification string, one which uniquely matched the single usage of "of" that was to be changed.)

Next, to verify that the whole segment is correct, our typist moved the pointer back to above the top of the segment with the t request on line 16, and then he asked the editor to print (with the p request) the next 1000 lines of his segment. Although he knew that his segment did not contain 1000 lines, he did not want to count them; when the user asks for a larger number than necessary, the editor merely prints to the end of the segment, then stops. Thus, we have the final segment contents printed on lines 19-23. The comment "No line." on line 18 is inserted whenever the pointer is not pointing at a line; for example, when it is pointing to the top of the segment. Similarly, the comment EOF on line 24 is printed whenever any request causes the pointer to run past the end of the segment. Our typist then typed the request w (write) on line 25, which means "put the segment away in the storage system". Being finished with the editor he then typed q, for quit. The editor responded by returning to command level, as shown by the ready message on line 27.

To illustrate the ability of the editor to modify a segment, lines 29 through 83 are a typical editing session. In this session, the typist made some changes to the segment containing the poem that had been typed in before.

The typist started from command level, just as before, typing the name of the editor and the name of the segment to be edited. This time, since the segment already existed, the editor began in edit mode rather than input mode. The typist wanted to add a line following the last line, so he had to move the pointer to the last line. Noticing that the last line contained the word "tiger", on line 31 he typed a request to locate that string of characters. Now it becomes apparent why the editor always prints the line it has moved the pointer to, as on line 32 -- there were two lines containing the word "tiger", and the editor had located the first one. The typist should have used the request:

```
l the tiger
```

on line 31, since only the last line contains the string of characters "the tiger". Seeing his mistake, the typist took advantage of a special convention: if he types a locate request with no character string, the previous locate request will be repeated, with the effect in this case that the next instance of the string "tiger" will be located. This he did on line 33, and the editor responded on line 34 with the last line of the segment. Then, using the i (insert) request, which inserts a line after the pointer, our typist on line 35 added a single line to the end of the segment.

Next, he decided that his poem needed a heading, so he moved the pointer back to the top of the segment with the t (top) request on line 36. Since the heading is to be more than one line, he decided to switch temporarily to input mode by typing the mode-switch character, a line containing only a single period, on line 37. He followed this with two lines to be stored in the segment following the current pointer position (which in this case was at the top of the segment). Note that line 40 is completely blank--presumably the typist wanted a blank line in his segment at that point. Having now finished typing the new material, the typist switched back to editing mode, went back to the top of the segment, and on line 44 requested that it be printed. As we see on lines 45-54, the segment appeared as before, except for the three added lines, two at the start and one at the end.

Next, our typist exhibited one of the most powerful features of this editor, its multiline change request. On line 56, he requested that the string "tiger" be replaced by the string "giraffe" everywhere it appeared on the next 1000 lines following the pointer. Thus, every occurrence of "tiger" in the entire segment was sought out and changed by the editor. For verification, the editor printed each changed line (lines 57 and 58), and then reported that it encountered the end of the segment (line 59). Finally, the typist decided that the line saying "anonymous" was superfluous, so he first moved the pointer to it (lines 60 and 61), and then deleted it (line 63). Finally, he wrote out the resulting edited segment, and then asked the editor to return to command level.

As an independent check on the contents of the resulting edited segment, he then typed the print command, as shown on line 68. This library program will print any text segment; first it prints a header giving the segment's name and the date and time (line 70), then it prints the contents of the segment.

With this brief introduction, the next steps to familiarity with the editor are to read the edm command write-up in the Reference Guide, and then to type in and edit a small segment of your own.

Some pointers:

1. It is useful to remember that the editor makes all changes on a copy of the segment, not on the original. Only when you issue a w (write) request does the editor overwrite your original segment with the edited version. If the user types q (quit) without a preceding w (write), the editor warns him that editing will be lost and the original segment will be unchanged, and gives him the option of aborting the request.

2. Don't ever press the quit button while in the editor, unless you are prepared to lose all of the work you have done since the last w (write) request. If you press quit while a w request is in progress, you may even damage the original version of the segment.

3. If one has a lot of typing or editing to do, it is wisest to occasionally (say every 10-15 minutes) issue a w request, to insure that all the work up to that time is permanently recorded. Then, if some accident should occur (e.g., a system failure, or the telephone line disconnects), you will lose work only back to the last w request.

4. Some requests are more expensive in computer resources than others. In particular, frequent movement of the pointer back to the top of the segment should be avoided. If possible, it is best to plan ahead, and try to do as much editing as possible with a single pass of the pointer through the segment. The larger the segment, the more important this consideration becomes.

5. The request to move the pointer backward, while very handy, is very expensive to use, since the editor actually has to move the pointer to the bottom, then back to the top, then to the correct location.

6. Be sure that you have switched from input mode to edit mode before typing editing requests, including the requests to write and quit. If you forget, the editing requests will be stored in your segment, instead of being acted upon. You will then have to locate and delete them.

7. The only frequently-used requests which have not been illustrated are the next (n) and backup (-) requests. The remaining requests are less important and you can safely ignore them to start with.

8. As one becomes more and more familiar with the use of edm, he may conclude that it provides verification responses more often than necessary, thus slowing him down. The requests v and k are used to control the editor's verbosity. At about the point where one feels confident enough to use these two requests constructively, it is probably time to begin studying the more sophisticated editor, qedx. The qedx editor provides the user with a repertoire of more concise and powerful requests, which permit more rapid work.

Using the Multics Storage System

In the previous section we saw how a text segment may be created and edited. In this section, we will explore some of the features of the system which allow such segments to be organized and stored for later use.

The user in our last example chose the name poem for his segment. Multics tries to allow the user as much flexibility as possible in choosing names for segments. Since the system has many users, who may be strangers to one another, this need for flexibility suggests that the segments belonging to any one user be grouped in such a way that he can choose names without worry that some other user has already used that name. This grouping is accomplished by an entity known as a _directory_. A directory may be conveniently thought of as a segment containing a list of names of other segments.*

Typically, each user has a directory for his own segments. Within a single directory, each segment must have a different name, but two different directories may contain segments with the same name. By a simple extension of this convention, directories are also given names, so a user's directory may contain the names not only of his segments, but also of additional directories he has created. These additional directories may contain the names of more segments. When a directory name is found in a directory, it is said to be an _inferior_ directory; the naming directory is said to be _superior_ to it. A user's motives for putting some of his segments in inferior directories may be several:

- He may have two segments to which he wants to give the same name; they must not be in the same directory.
- He may have many segments, and would like to keep them grouped by category. As we shall see, he can ask for a list of all the names in any one directory, and thus in one of his categories.
- He may wish to protect a certain group of segments all in the same way; when he creates a new such segment, he can protect it the same way as the others by putting it in the appropriate directory; he need not think through the protection specification again.

---

* Although a segment is technically only _named_ by a directory, it is common terminology to refer to a segment as being _stored_ in a directory. Of course, the segment is actually stored on some disk or drum storage device; only its location on that device is stored in the directory. This distinction is important in the case of _links_, which name segments stored in other directories, rather than providing for their storage directly.

- Whenever a program asks for a segment by name, a search is undertaken for the segment. This search is controlled by specifying a list of directory names. Thus, he may create several directories in order to arrange that the search proceed in a fashion he prefers.

It should be clear, then, that the concept of a directory is a key to several different features of the Multics storage system. The idea of superior and inferior directories is extended by the requirement that all the directories in the system together form a hierarchy, or tree. The directory at the base of the tree, which is superior to every directory of the system, is called the root directory.

Figure 3-3 is a typical directory arrangement. The root directory in that example contains two entries, both of which are names of other directories. One of these two directories contains the library of system programs, while the other, named udd (for user_directory_directory) contains one entry for every user of the system, namely Smith and Jones. These two users each have a directory with their names on it, and in addition, Smith has chosen to add another directory inferior to his own, named old_dir; he has placed three segments named x, y, and z in old_dir.

Whenever a Multics program wishes to read or change the contents of a segment, it is required to specify the name of the segment it wants. Every segment has a path name which is formed as follows: trace the directory structure down from the root to the desired segment, writing in order the name of every directory on the path, and finally the name of the segment itself. Now, concatenate all these names into a single long name, placing the "greater than" character between the individual names. Thus, the path name of the edm command, found in the library, would be

```
root>library>edm
```

By convention, since every path name would begin with the letters "root", these letters are left off, so one would use the path name

```
>library>edm
```

to refer to the edm command. Similarly, Jones' segment named lp.pl1 has the path name

```
>udd>Jones>lp.pl1
```

and Smith's segment named x has the path name

```
>udd>Smith>old_dir>x
```

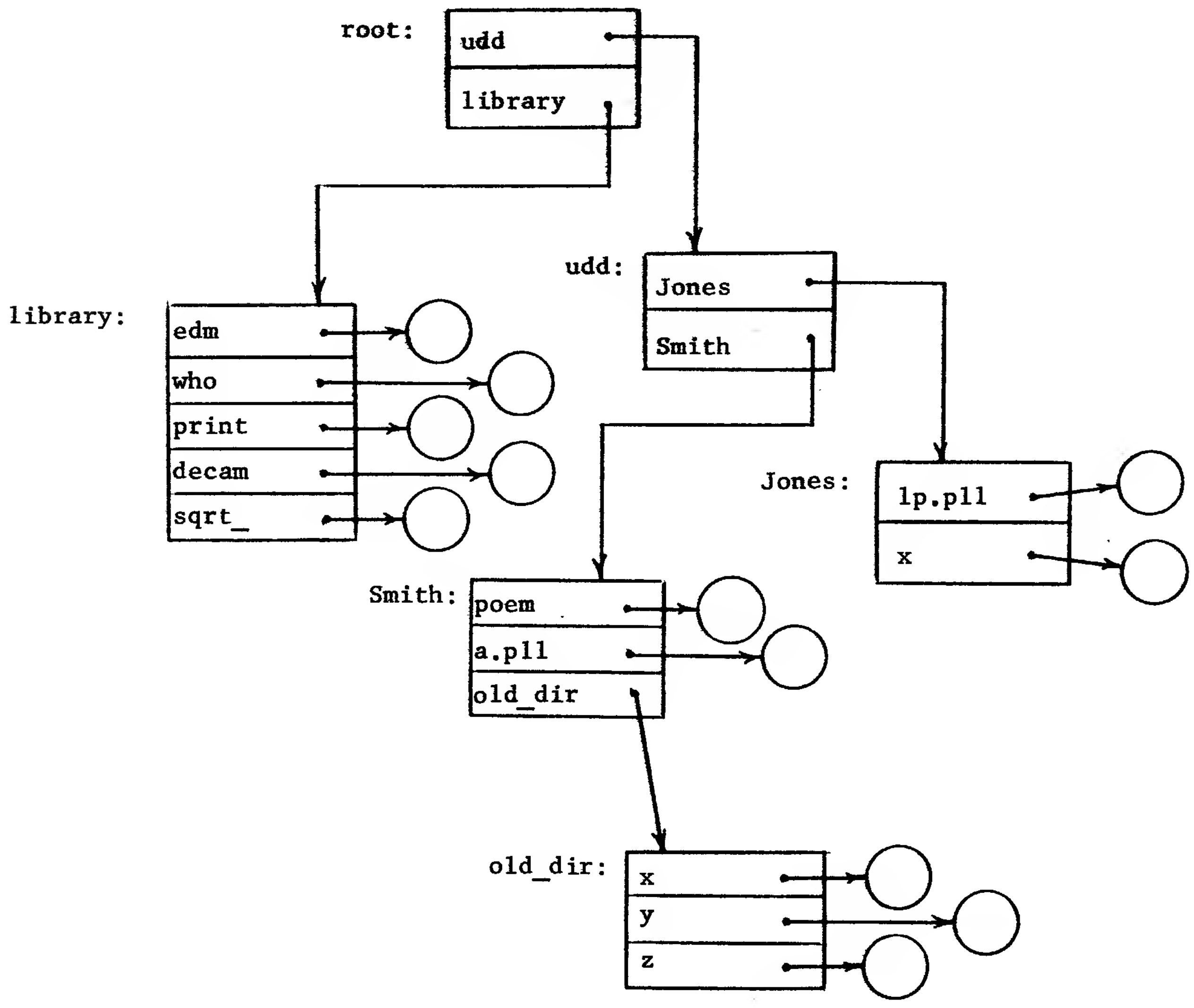


Figure 3-3: Typical Multics Directory Hierarchy. Directories are rectangles; segments are circles.

which is clearly distinct from Jones' segment x, which has the path name

    >udd>Jones>x

To avoid the need for typing full path names, which may not be easily remembered (or even known, in some cases), the system remembers for each logged in user the path name of one directory in which his activity is centered: his working directory. All names which do not begin with a "greater than" sign are considered to be relative to his working directory. Thus, for example, Smith might choose as his working directory the path name

    >udd>Smith

in which case when he uses the name

    poem

he will be referring to the segment with path name

    >udd>Smith>poem

and when he uses the name

    old_dir>x

he is referring to the segment with path name

    >udd>Smith>old_dir>x

The system automatically chooses an initial working directory for a user when he logs in, but he is free to change the path name of his working directory to any other directory in the system. He makes this change by invoking one of several commands used for interaction with the storage system. As before, it is easiest to understand these commands by following a series of sample scripts, which are based on the directory organization illustrated in Figure 3-3. Suppose that Jones has logged in, and the system has assigned him the directory

    >udd>Jones

as his working directory to start with. (The script may be found in Figure 3-4.)

On line 1, he typed the command print_wdir, which merely prints the path name of his current working directory on line 2. (This command is quite handy if one forgets where he is, or needs confirmation that he typed his last command to change directories correctly.) Next, on line 5, he typed the list command, which prints the contents of the working directory. On line 7 the list command printed a summary of the directory contents. Jones'

```
1*     print_wdir
2      >udd>Jones
3      r  1210  .137  .812  27
4
5*     list
6
7      Segments = 2, Records = 4.
8
9      r w   1  lp.pl1
10     re    3  x
11
12     r  1212  .216  1.762  33
13
14*    create foo
15     r  1213  .320  3.728  77
16
17*    list
18
19     Segments = 3, Records = 4.
20
21     r w   0  foo
22     r w   1  lp.pl1
23     re    3  x
24
25     r  1215  .202  1.856  49
26
27*    createdir  mypoems
28     r  1216  .151  1.482  0
29
30*    change_wdir  mypoems
31     r 1218  .089  .306  17
32
33*    print_wdir
34     >udd>Jones>mypoems
35     r  1219  .119  .056  14
36
37*    list
38     directory empty
39     r  1219  .147  1.406  42
40
41*    copy >udd>Smith>poem  limerick
42     r 1220  .311  1.732  53
43
44*    list
45     Segments = 1, Records = 1.
46
47     r w   1 limerick
48
49     r  1220  .219  2.162  41
50
51*    change_wdir >udd
52     r  1221  .067  .646  30
53
```

Figure 3-4: Example of Use of the Multics Storage System.

```
54*   list  -a
55
56    segments = 0
57
58    links = 0
59
60    Directories = 2, Records = 24
61
62    s    1 Smith
63    sma  1 Jones
64
65    r  1222  .077  .305  9
66
67*   cwd  Smith>old_dir
68    r  1222  .138  1.406  62
69
70*   status x
71
72    names:          x
73    type:                    segment
74    date used:               [illegible]
75    date modified:           [illegible]
76    branch modified:         [illegible]
77    bit count:               [illegible]
78    records used             1
79    mode                     rw
80
81    r  1223  .439  3.402  62
82
83*   change_wdir
84    r  1224  .111  1.130  41
85
86*   link  >udd>Smith>old_dir>x Smithx
87    r  1225  .178  1.700  41
88
89*   listnames -a
90
91    Segments = 3, Records = 4.
92
93    foo
94    lp.pl1
95    x
96
97    Directories = 1, Records = 1.
98
99    mypoems
100
101   Links = 1.
102
103   Smithx
104
105   r  1227  .626  2.154  43
106
```

Figure 3-4 (continued)

directory (refer to Figure 3-3) contained only two entries, and these segments occupied a total of four records, the unit of storage space. One record has room for up to 4096 printed characters, or 1024 computer words.

Starting on line 9 is the three-column list of names of segments in this directory. Working back from the right, the third column is the segment name (32 characters or fewer in length), the second column is the number of storage records occupied by this segment, and the first column tells the mode of access this user is permitted to this segment. Up to three letters may appear in this column, each letter indicating an additional privilege:

r (read)	The user may read the contents of this segment.
e (execute)	The user may run this segment as a program.
w (write)	The user may rewrite the contents of the segment.

We will return later to the subject of setting these access mode indicators. For the moment, we will merely observe that they exist, that different users may have different access mode indicators for the same segment, and that the system enforces the access mode restrictions.

On line 9 is listed a segment which has a "period" as part of its name. In general, the storage system is happy to allow any character except the "greater than" sign in a segment name. The user of the storage system may wish to attach some special meaning to some character, and one such system-wide convention is illustrated on line 9: a segment name may consist of components, separated by periods. As far as the storage system is concerned, the name is one long string of letters with interspersed periods; the user by convention attaches meaning to the components. It is customary, for example, for source language programs to be given a two-component name. The first component is chosen by the user, and the second component is the name of the source language. Thus, the name lp.pl1 is evidently attached to a program written in the PL/I language.

On line 14, the user typed a command which creates a new segment, and upon reissuing the list command on line 17, we see the newly created segment included in the listing. Note that the create command attached an access mode indicator of "r w". Note also that since no information has been written in the segment yet, its space occupied is 0.

On line 27, the user created a directory inferior to his own, named mypoems, and on line 30 he typed the command which changes his working directory to the new inferior directory. As a check, on line 33 he asked to print the name of his working directory, which is now

```
>udd>Jones>mypoems
```

When he tried to list the contents of his new directory, on line 37, he received an appropriate error comment.

To illustrate a typical use of segment names, on line 41 he typed a copy command. The copy command works as follows:

```
copy a b
```

- The segment named a is located in the hierarchy.
- A segment named b is created.
- The contents of a are copied into b.

Both the names a and b are subjected to the conventions about working directories. Thus, on line 41, the name a is

```
>udd>Smith>poem
```

which, since it begins with the "greater than" character, is taken to be a full path name and requires no interpretation. The name b is

```
limerick
```

which, not starting with the "greater than" character, must be interpreted relative to the current working directory. Thus, name b for this case is taken to be

```
>udd>Jones>mypoems>limerick
```

A segment of that path name was thus created, and the contents of Smith's poem were copied into it. To prove this, the user next typed "list", and found one segment, named limerick, in his new directory. Its size was nonzero, so something must have been written into it by the copy command.

We should pause at this moment to observe that copying of segments is the exception, rather than the rule, in Multics. Normally several different users will share the same copy of a segment, either by giving the full path name when they wish to access it, or by placing in their working directory a link to the segment. Copying is performed only if one wishes to make a modification to a segment, but keep the original version also.

Continuing our example, on line 51, the user began exploring the rest of the directory structure by typing commands to change his working directory to one higher in the directory hierarchy. He then on line 54 listed the contents of this directory.

The list command presumes that most often one wants only a list of segments, not of inferior directories, so it normally does not print directory names. If the argument -a (for all) is given to the list command, it will list everything in the directory, not just segments. Thus, on lines 56-63, we see the summary of contents, and the names of the two directories inferior to udd. Note that Jones has more access to his own directory than he does to Smith's. If Smith were to try this same experiment, he would probably find that he has more access to his own directory than he has to Jones'. Access modes for directories are described below under Access Control in Multics.

Next, on line 67, Jones switched his working directory into Smith's own inferior directory, and used the status command to find out all he could about segment x.

Finally, he returned his working directory to the place where he started, by typing the command change_wdir with no arguments. The change_wdir command has tucked away the name of his original working directory to allow such a move to be specified easily, since it is very common.

Next, the user placed in his directory a link to Smith's segment x, as referred to above. Note that one can make a link to another directory, if desired, also. This feature allows one to talk about any entry in that directory with a name briefer than the path name from the root.

Finally, he listed just the names of everything in his directory. Figure 3-5 illustrates the modified directory structure.

While the sample scripts described here are useful for getting a flavor of how the system is typically used, much additional insight can be gained by experimenting with the system itself. For example, the following series of experiments is suggested:

1. Log in

2. Print the name of your working directory with the print_wdir command.

3. List the contents of your working directory with the command "list -a".

4. Switch to the directory immediately superior to yours with the change_wdir command. Give as the name of the directory to switch to, the name printed in step 2, with the last component stripped off.

5. Repeat steps 2-4 until you have reached the root directory. (To enter the root directory, use a "greater than" sign for its name.)

6. Explore downward from the root to see how far you can go into other parts of the directory hierarchy.

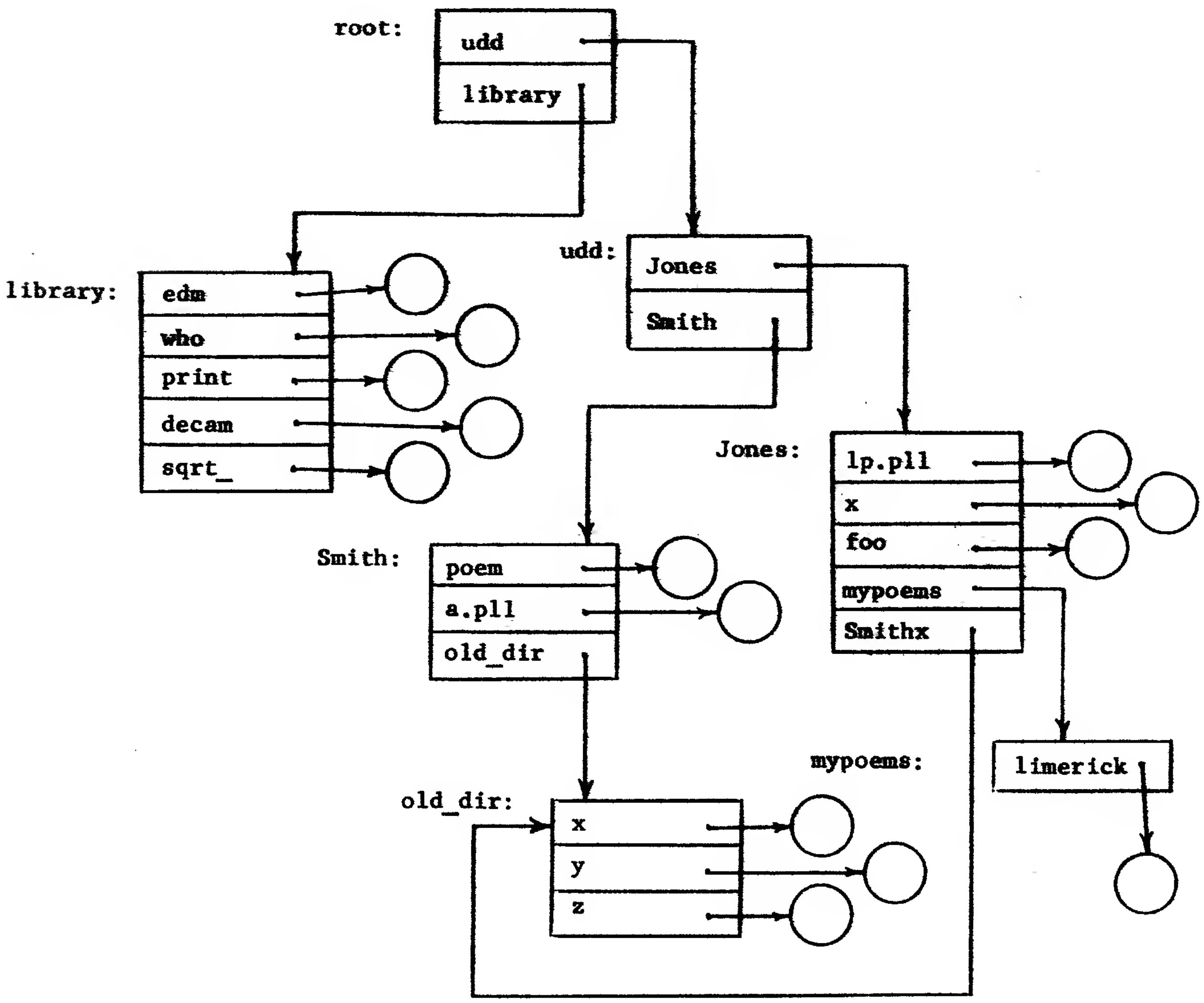


Figure 3-5: Directory Hierarchy of Figure 3-3 (After Manipulation by Example Script). Directories are rectangles; segments are circles.

Finally, we have not yet mentioned three commonly used convenience features of the Multics storage system:

1. Any time a segment name must be typed, one may specify either the path name from the root, or a relative path name starting from the current working directory. We have already seen two examples of this feature above, in typing names of segments located below the working directory. One can also give relative path names for segments not below the working directory by typing an initial "less than" sign for each level up in the hierarchy needed to get to the segment in question. Thus, if the working directory is

   ```
   >udd>Smith
   ```

   Then the relative path name

   ```
   <Jones>lp.pl1
   ```

   is taken to mean

   ```
   >udd>Jones>lp.pl1
   ```

2. Any segment, link, or directory may have several names, if desired. The addname command is used in this connection. Multiple names are handy in cases where a new name is wanted, but some programs (or users) still use the old one. Also, a segment with a long name may be given a second, shorter name for typing convenience.

3. There are conventions for talking about groups of segments with similar names, using an asterisk to specify the parts of the name that vary within the group. Thus, the command

   ```
   list *.pl1
   ```

   would list all segments in the current working directory which have two-component names ending with .pl1.

More details on these three features, as well as many other storage system features and options which are less commonly exercised, may be found in the MPM Reference Guide sections on Using the Multics Storage System, and the MPM Reference Guide section, Constructing and Interpreting Names.

Access Control in Multics

In the examples given above, each segment had an access mode which indicated the user's ability to read or write in a given segment. The access modes are not universal; Multics permits different users to have different access modes for the same segment. Further, careful control is maintained over who may set or change the access mode of a segment. These facilities permit control of privacy of information in a large variety of ways. Multics contains some very powerful features for controlling access which allow construction of restricted access general-purpose subsystems by users with no special privileges. Though he may not immediately see a use for the fully sophisticated mechanism, the casual user should be familiar with some of the more routine aspects of access control.

The most important piece of the access control mechanism is the _access control list_, abbreviated ACL. Every segment has its own ACL. An ACL consists of a list of names of users who are permitted to use a segment, along with the modes (read, execute, or write) which they may use. To make ACLs meaningful, every user of Multics is registered, which means a standard name, different from everyone else, is recorded for him. The password, typed at login time, is a check on the authenticity of a user claiming that he is registered. For convenience in specifying access control, users may be organized into groups who are working together. Each such group is given a unique name also, known as a _project identifier_. For purposes of controlling access, the name of a logged in user is the concatenation of the user's registered name and his project's name. Two typical access control names are:

```
Williams.Apollo.a
Jones.MathSim.a
```

The third component of the name can be different for each instance of a particular user, if he has two jobs in the system at once, or is logged in twice. An ACL consists of a series of access control names, followed by the mode of access allowed to that name. A user can access a segment only if his name matches one of the entries on the ACL. For example, the ACL

```
Williams.Apollo.a      re
Jones.MathSim.a        rw
```

would grant access to just those two users, and no one else. To grant access to all members of a given project, one of the ACL entries may specify _anyone_ by placing an asterisk in the field normally occupied by the personal name. Similarly, asterisks may be placed in the other two fields, Thus the access control list

```
Williams.Apollo.*      rew
*.Apollo.*             rw
*.*.*                  r
```

would permit Williams, when working on project Apollo, to access the segment with all modes of access, all other Apollo project members with slightly restricted access, and all other users of the system, with read access only.

Access control lists are constructed and modified with the aid of three commands: setacl, deleteacl, and listacl. Permission to use these commands is based on a simple hierarchical rule: directories also have access control lists. Permission to modify a directory carries with it the permission to set the ACLs of segments stored in that directory. Thus, most users are assigned a directory by their project supervisor; he sets the ACL of the directory to allow the user to modify the directory, and the user then has complete control over who may access segments he places there.

One minor point of interest here is that the project supervisor must have had permission to modify the next higher level directory in order to create the user's new directory, as well as to set the ACL permitting the user to modify the new directory. That permission is derived in the same way, by an ACL controlling the next higher directory. This general pattern continues up to the root directory, which has an ACL which permits only the system administrator ability to modify its contents.

Multics distinguishes among several ways of using directories, and an ACL intended for a directory indicates these ways in a manner analogous to the access modes of a segment. The directory access modes are:

s (<u>s</u>tatus)	The user may list the contents and find out the attributes (such as ACLs) of the entries in the directory.
m (<u>m</u>odify)	The user may delete entries from the directory and may modify the attributes of entries in the directory.
a (<u>a</u>ppend)	The user may add an entry to the directory, but he may not later delete it unless he also <u>m</u> access.

The "a" access mode is handy for implementing mailbox facilities in which the only form of access is to leave a message.

In order that the user not be plagued with constant need to specify ACLs, each directory contains an initial access control list (inital ACL) which is automatically placed on every entry added to that directory. Also, most standard facilities for creating segments routinely specify appropriate access for at least the user who created the segment. Thus, a common strategy is to place in the inital ACL the entries

```
*.*.* re
*.*.* s
```

thus allowing all other users freedom to explore, but not change, the segments and directories contained in the user's directory.

Finally, certain system services such as off-line printing of segments and backup copying of new and modified segments are performed by system processes which must have access to any segments they print or copy. Appropriate ACL entries are automatically placed on every segment unless the user takes explicit steps to prevent them from appearing.

Where to Go from Here

This chapter has illustrated the typical usage of some commonly used commands. However, even a beginning user will rapidly develop needs for many of the more sophisticated facilities available. On the other hand, a cover-to-cover reading of the Reference Guide is probably not the most efficient method of gradually expanding one's grasp of system facilities. Reading the following sequence of material from the Reference Guide may be useful in getting started:

1. Read the Reference Guide section entitled The Multics Command Repertoire to become familiar with the kinds of commands available, and their names.

2. Peruse the remaining parts of Section 1 of the Reference Guide (The Multics Command Language Environment) so that you will know what kinds of questions are answered there. Detailed study of these parts can be deferred to the time when a need arises.

3. Read the Reference Guide section, The Storage System Directory Hierarchy, and skim the remainder of the sections on Using the Multics Storage System.

4. Read the following command descriptions; they represent the set which will be most used, at first:

```
edm          link          login
print        unlink        logout
dprint       list          rename
delete       listacl       pl1
help         setacl        getquota
             mail          who
```

5. Read the first few pages of the description of the debug command. This facility is extremely powerful, but a beginner will find that there are a lot of ideas to master before he can use debug to its full effectiveness.

6. Read Chapter Four for an introduction to the programming enivironment.

7. Look at the Reference Guide section, List of System Status Codes and Meanings, to see what kinds of information are listed there.

8. At the next level down, the following less frequently used commands are also good to know about:

```
copy                 change_wdir
hold                 print_wdir
start                archive
new_proc             status
release              where
program_interrupt
```

9. Before beginning to write programs in earnest, review the section on The Multics Programming Environment, and especially the part entitled The Subroutine Repertoire.

10. Finally, read the section on Use of the Input and Output Facilities.

The set of section and command write-ups suggested above should provide a thorough introduction to both the facilities available on Multics and also the kinds of reference material found in this manual.

CHAPTER 4

PROGRAMMING IN THE MULTICS ENVIRONMENT

September 20, 1973

A programmer may, if he wishes, treat Multics as simply a PL/I, FORTRAN, APL, BASIC, or LISP machine, and contain his activities to just the features provided in his preferred programming language. On the other hand, much of the richness of the Multics programming environment involves use of system facilities for which there are no available constructs in the usual languages. To use these features, it is generally necessary to call upon library and supervisor subroutines. Unfortunately, a simple description of how to call a subroutine may give little clue to how it is intended to be used. The purpose of this chapter is to illustrate typical ways in which one utilizes many of the properties of the Multics programming environment.

The programmer choosing a language for his implementation should carefully consider the extent to which he will want to go beyond his language and use system facilities of Multics which are missing from his language. As a general rule, one may say that each of the Multics languages matches some well-known standard for completeness of that language (e.g., .RSI or IBM). However, in going beyond the standard languages, the programmer will find that Multics tends to be biased towards convenience of the PL/I programmer. For example, if one plans to write programs which directly call the Multics storage system privacy and protection entries, he will be asked to supply arguments which are, in PL/I, structures. If he is writing in FORTRAN or BASIC, he has no convenient way to express such structures. Note that the situation is not hopeless, however. Programs which stay within the original language can be written with no trouble. Also, in many cases, one can construct a trivial PL/I interface subroutine, callable from, say, a FORTRAN program and which goes on to reinterpret arguments and invoke the Multics facility desired. Using such techniques, almost any program originally prepared for another system can be moved into the Multics environment.

Probably the quickest way for an experienced programmer to get a feel for how to program in a new environment is to examine sample programs. This chapter consists of several examples of programming for Multics. Each program is annotated with comments to guide the reader. Unfortunately, programs do not always invoke features in the best order for understanding, so the following strategy may be useful: as you read each comment, if its implications are clear and you feel you understand it, check it off. If you encounter one which does not fit in to your mental image of what is going on, skip it for the moment. Later comments may shed some light on the situation, as will later reference to other parts of the MPM. Finally, a hard core of obscure points may remain unexplained, in which case the advice of an experienced Multics programmer is probably needed. Be warned that the range of comments is very wide, from trivial to significant, from simple to sophisticated, and from obvious to extremely subtle.

The notes presume that the reader is familiar with the PL/I language. Only those aspects of the language for which Multics provides some unusual implication are mentioned. The programs have been printed out on an IBM 2741 (golf-ball) typewriter, so the ASCII circumflex character appears as a hooked overbar.

Finally, some comments provide suggestions for "good programming practice." Such suggestions are usually subjective, and often controversial. Nonetheless, the concept of choosing among various possible implementation methods one which has clarity, is consistent, and minimizes side effects is valuable, so the suggestions are provided as a starting point for the reader who may wish to develop his own style of good programming practice.

Basic Addressing Techniques

The most significant difference between the Multics programming environment and that of most other contemporary computer programming systems lies in its approach to addressing online storage. Most computer systems have two sharply distinct environments: a resident file storage system in which programs are created, and translated programs and data are stored, and an execution environment consisting of a processor (actually allocated in short time bursts) and a "core image", which contains the instructions and data for the processor. Supervisor procedures provide subroutines for physically moving copies of programs and data back and forth between the two environments.

In Multics, the line between these two environments has been deliberately blurred, so as to simplify program construction: most programs need to be cognizant of only one environment rather than two. This blending of the two environments is accomplished by extending the processor/core-image environment. In Multics, the share of the processor is termed a _process_, and the core image is abstracted into what is called an _address space_. Each user when he logs in is assigned one newly created address space, and a single process which can execute in it.

A Multics address space is not like the usual core image, however: it is larger, and it is segmented*. A segment may be of any size between 0 and 256K 36-bit words and an address space may have a large number of segments -- a typical Multics process has about 200 segments. (The hardware places a limit of 256K distinct segments, but table sizes in the current software limit an address space to a number closer to 2000.) Typically, each separately translated program resides in a different segment; collections of data which are large enough to be worthy of a separate name are placed in a segment by themselves.

The segment is also the unit of storage of the Multics catalogued file storage environment. (Called the _Multics storage system_.) These two environments, distinct in many other systems, are automatically mapped together on demand, by the Multics virtual memory system. When a program already appearing in the current address space calls to another one which is not yet there, a _dynamic linking fault_ occurs, the supervisor locates the needed procedure, and maps it into the current address space, assigning it some as yet unused segment number. Similarly, data segments are mapped into the address space. In contrast to many other systems, this address space is retained throughout the login session, and its contents gradually are increased as different programs and data objects are accessed. (Facilities are also available for starting over with a new address space, or removing items no longer needed in the address space.) Finally, all supervisor procedures and commands called by the user are mapped into the very same address space. Thus, there is a great uniformity of access methods, to user-written programs, to data, to library or supervisor programs, and to items never before used but catalogued in the storage system.

As will be seen in the examples which follow, the effect of the mapping together of these two environments can range from the negligible (programs can be written as though there were a traditional two-environment system, if desired) to a significant simplification of programs which make extensive use of the

---

* This discussion presumes that the reader is familiar with the purposes of and mechanisms which allow memory segmentation. For further background in this area, see the bibliography at the end of Chapter One and the first parts of Chapter Two. In addition, books by Organick (_The Multics System: an Explanation of its Structure_) and Watson (_Time Sharing System Design Concepts_) motivate segmentation.

catalogued storage system. We begin with seven brief examples of programs which are generally simpler than those encountered in practice, but which illustrate ways in which on-line storage is accessed in Multics.

1. Internal Automatic Variables. The following program types the word "Hello" on four successive lines of terminal output:

```
a:      procedure;
        declare i fixed binary;
        do i = 1 to 4;
             put list ("Hello");
             put skip;
             end;
        return;
        end a;
```

The variable i is by default of PL/I storage class "internal automatic": in Multics it is stored in the stack of the current process and is available by name only to program "a" and only until "a" returns to its caller. It is declared binary for clarity, so that there will be no question in the reader's mind whether or not a presumably slower decimal addition is involved.

2. Internal Static Variables. The following program, each time it is called, types out the number of times it has been called:

```
b:      procedure;
        declare j fixed binary internal static initial(1);
        put list (j, "calls to b.");
        put skip;
        j = j + 1;
        return;
        end b;
```

The variable j is of PL/I storage class "internal static"; in Multics it is stored in b's linkage section (discussed later) and is available by name only to program b. Its value is preserved for the life of the process, or until procedure b is recompiled, whichever time is shorter. The "initial" declaration causes the value of j to be initialized at the time this procedure is first used in a process.

3 and 4. External Static. Suppose we wish to set a value from one program and have it printed by some other program in the same process:

```
c:      procedure;
        declare z fixed binary external static;
        z = 4;
        return;
        end c;


d:      procedure;
        declare z fixed binary external static;
        put list (z);
        put skip;
        return;
        end d;
```

In both programs, the variable z is of PL/I storage class "external static"; in Multics it is stored in a particular segment (named stat_ by default, but changeable), and is available to all procedures in a particular process, until the process is destroyed. External static is analogous to COMMON in FORTRAN, but with the important difference that data items are accessed by name rather than by relative position in a declaration.

Each variable which is accessed in this form generates a dynamic linking fault the first time it is used. Later references to the variable by the same procedure on that or subsequent calls do not generate the fault. A more complete discussion of dynamic linking appears in a later section of this chapter.

5. Direct intersegment References. The following program prints the sum of the 1000 integers stored in the segment w:

```
 1      e:      procedure;
 2              declare w$(1000) fixed binary external;
 3              declare (i, sum) fixed binary;
 4              sum = 0;
 5              do i = 1 to 1000;
 6                   sum = sum + w$(i);
 7                   end;
 8              put list (sum);
 9              put skip;
10              return;
11              end e;
```

The dollar sign is recognized as a special identifier by the PL/I compiler, and code for statement 6 is constructed which anticipates dynamic linking to the segment named w. Upon first execution, a dynamic linking fault is triggered, and a search undertaken for a segment named w. If one is found, the link is "snapped," which means that all future references will occur with a single machine instruction.

If no segment named w is found, the dynamic linker will return to command level and report an error to the user. As described later, it is possible to create an appropriate segment named w, and then continue execution of the interrupted program, if such action is appropriate.

6. Reference to Named Offsets. The following procedure calculates the sum of 1000 integers stored in segment x starting at the named offset u:

```
f:      procedure;
        declare x$u(1000) fixed binary external;
        declare (i, sum) fixed binary;
        sum = 0;
        do i = 1 to 1000;
             sum = sum + x$u(i);
             end;
        put list (sum);
        put skip;
        return;
        end f;
```

The difference between this example and the previous one is that segment x is presumed to have some substructure, with named internal locations, called offsets. To initially create a segment with such a substructure, one normally uses one of the compilers or assemblers, since an inbound linkage section must be constructed for the segment to indicate to the linker where within the segment the offsets may be found. Unfortunately, the PL/I language permits specification of such structured segments only for procedures, not for data. The ALM assembler can be used for creating structured data segments. (It is expected that in the future better techniques will become available.)

7. External Reference Starting With a Character String. In many cases, one starts with a character string representation of the name of a segment which is to be accessed. In those cases, a call to the Multics storage system is required in order to map the segment into the virtual memory and to obtain a pointer to it:

```
g:      procedure(string);
        declare string character(*);
        declare p pointer;
        declare (i, sum) fixed binary;
        declare v(1000) fixed binary based(p);
        call hcs_$make_ptr (string, p);
        sum = 0;
        do i = 1 to 1000;
             sum = sum + v(i);
             end;
        return;
        end g;
```

The calling sequence to hcs_$make_ptr is simplified from real life. The real calling sequence requires specification of several options unimportant to us here. (This is the only sample program which will not work if typed in literally as shown. See the write-up of hcs_$make_ptr in the subroutine section of the MPM for the complete calling sequence.)

One may also use, in place of hcs_$make_ptr, another storage system entry named hcs_$initiate. When using hcs_$initiate, one directly specifies the path name of the segment desired: no search is undertaken for the segment as in the case of a dynamic linking fault. This procedure differs greatly from the examples above, in which a search is involved. An intermediate situation, in which library routines are used to construct a tree name starting with an entry name, is found in the "simple text editor" example, which appears later in this chapter.

A Program Which Tests for Prime Numbers

In figure 4-1 is a typical small PL/I program, which may be used as a model for many simple calculations not involving special Multics system properties. The program is confined entirely to the PL/I language; presumably it would run unchanged on any computer system which has a PL/I, assuming that all the necessary PL/I features are available. The program is organized assuming that input and output will go from and to an interactive console. The comments following are keyed to the line numbers printed to the left of the program. (Note: the source program is typed in <u>without</u> line numbers. We have added them here to facilitate making comments, with an asterisk indicating lines typed by the user, as in chapter 3.)

line comment

5. All identifiers are explicitly declared, to be sure that no suprise defaults occur, and to make easier the job of reading the program for someone else who is asked to maintain it.

7. These two identifiers are not explicitly used in the program, but they are implicitly involved in the put list and get list statements.

9. Character and bit strings are delimited with the ASCII double quote mark in the Multics PL/I language.

9. Note that the upper case and lower case letters are different, whether appearing in comments, literal strings, or identifiers.

13. The underscored word <u>not</u> will properly go through all the mechanisms and come out the other end. If we had used edit-type I/O statements (that is, format statements) we would have noticed one minor problem: the character

```
 1*      print primetest.pl1
 2
 3       primetest:          procedure;
 4
 5       declare prime_input fixed binary;
 6       declare (sqrt,mod) builtin;
 7       declare (sysprint,sysin) file;
 8
 9               put list ("Type prime to be tested:  ");
10               get list (prime_input);
11               if prime(prime_input)
12                    then put list (prime_input, "is a prime.");
13                    else put list (prime_input, "is not a prime.");
14               put skip;
15               return;
16
17       prime:  procedure(trial_prime) returns (bit(1));
18
19       declare trial_prime fixed binary,
20               trial_factor fixed binary,
21               last_factor fixed binary;
22
23               last_factor = sqrt(trial_prime);
24               do trial_factor = 2 to last_factor;
25                    if mod(trial_prime, trial_factor) = 0
26                         then return ("0"b);
27                    end;
28               return ("1"b);
29
30               end prime;
31
32               end primetest;
33
34       r 1406  1.712  9.359  176
35
36*      pl1   primetest
37       PL/I
38       r 1409  7.041  56.437  1217
39
40*      primetest
41(*)    Type prime to be tested:    121
42                           121          is not a prime.
43       r 1410  2.960  10.627  557
44
45*      primetest
46(*)    Type prime to be tested:    397
47                           397          is a prime.
48       r 1410  .305  3.172  98
```

Figure 4-1: A program which tests for prime numbers.

position counts in format statements are in terms of storage locations occupied by a character string rather than print positions required to print the character string. Thus the string <u>not</u> would require 9, rather than 3, spaces in a format specification. (Three letters, three backspaces, and three underscores.)

17. This internal procedure is not recursive, and meets several other rules which permit the compiler to generate a very fast (1-instruction) calling sequence to it. Storage for variables of the internal procedure is actually allocated in the automatic storage area of primetest itself for this special case. Thus, non-recursive internal procedures are quite economical organizing tools.

23. The algorithm used to test for primeness is actually quite brute force: the only work reduction technique it employs is to note that at least one factor of a number must be less than or equal to the square root of the number.

23. Note that the use of the sqrt built-in function involves conversion from integer to floating-point representation, and back. These conversions are automatically supplied by PL/I, but the programmer should be aware when he invokes them, so as not to trigger unnecessary conversion.

In the examples of use of the program, note that the ready message cost of use is substantially larger the first time the program is invoked. (Compare lines 43 and 48.) This effect is due to the initial dynamic linking of the procedure to its environment, including primarily the input and output mechanisms invoked by put and get.

<u>Checking on the Performance of a Program</u>

Often, after putting together a new program, one wishes to improve its performance. The simplest performance measuring tool available in Multics is to be found in the ready message. A slightly more sophisticated approach can be taken by using the "profile" option of the PL/I compiler. For example, if one wished to compile the primetest program using this option, he would proceed as in figure 4-2.

The numbers printed in the profile are statement-by-statement counts of the number of times that the statement was executed, and the number of machine language instructions which were involved. The latter number (in the column headed "COST") is shown as the sum of two parts, the inline instruction count, and the number of transfers out to PL/I support subroutines ("operators"). Thus, line 23 (containing a use of the single-precision fixed point modulo operator) was executed 30 times; it apparently consists of 13 machine language instructions, one of which is the call to the operator which performs the mod builtin function. The names in parentheses at

```
1*     pl1 primetest -profile
2      pl1, Version II
3      r 1605  9.089  40+758
4
5*     primetest
6(*)   type prime to be tested:      997
7                               997              is a prime.
8      r 1605  2.409  14+177
9
10*    print_profile primetest
11
12     LINE  STM  COUNT  COST  PROGRAM
13
14                             primetest
15       5    1     1     29
16       7    1     1      6 + 3  (stream_io  put_list_al  put_end)
17       8    1     1      7 + 3  (stream_io  get_list_al  get_end)
18       9    1     1     21 + 4  (stream_io  put_list_al  put_list_al  put_end)
19      12    1     1      7 + 2  (stream_io  put_end)
20      13    1     1      7 + 1  (return)
21      21    1     1     13 + 3  (fx1_to_fl2  call_ext_out  fl2_to_fx1)
22      22    1     1      7
23      23    1    30    390 + 30 (mod_fx1)
24      25    1    30    240
25
26     TOTAL                727 + 46
27     r 1606  1.703  4.991  151
```

Figure 4.2: Use of the execution profile feature.

the right are those of the operators involved. For example, line 21 of the program takes the square root of a fixed binary integer. Operator fx1_to_fl2 converts the integer to floating point representation for the square root routine. Operator call_ext_out performs the call to sqrt, and operator fl2_to_fx1 converts the result back to integer form.

Other performance measuring tools include the page_trace command, which prints out a list of recently-used pages. Various clock subroutines may be used to time the execution of subroutines to microsecond precision.

Debugging Programs on Multics

A variety of debugging tools are available on Multics. The most powerful of these is a program named debug, which permits source-language breakpoint debugging of PL/I and FORTRAN programs. The debug command also has many features useful to the machine language programmer, but we will concentrate here on a small subset of its features which can be quickly and easily applied to a PL/I program.

To understand the examples given below, one must first know a little about the Multics stack. The stack is essentially a push down list used to contain the return points from a series of outstanding interprocedure calls. It is also used for storage of automatic variables. If one were to stop a running program and trace its stack, he would find, starting at the oldest entry in the stack, a record of the procedures used to initialize the process, followed by the command language interpreter, followed by the procedure called at command level and any procedures it has called. If an unexpected error occurs (or the user presses the "Quit" button), the system will mark the stack at its current level, push it down, and call a new invocation of the command interpreter. Three special commands may then be invoked: release, hold, and start. If the user types release, the command interpreter will unwind the stack back to its own previous invocation, and discard the intervening stack contents. If the user types hold, the stack contents will be preserved indefinitely. If the user types start, the system will attempt to return to the interrupted computation to continue it. Depending on the nature of the error, and what the user has done since the error occurred, the restart attempt may or not succeed. The user may also type any other command, but upon completion of that command, the command interpreter will automatically perform a release operation, unless a hold has been requested. A common response to an unexpected error is to type hold, use other commands and debugging tools to discover and repair the error, and then type start, if it still makes sense to continue running the program.

Consider, now, the script of figure 4-3: The program printed on lines 3-11 scans the automatic array named "a", using illegal negative subscripts. Since the program does not specify

```
1*       print blowup.pl1
2
3        blowup:   procedure;
4
5        dcl       (j,a(10),loop_index) fixed binary;
6
7                  do loop_index = -1 to -100000 by -1;
8                       j = a(loop_index);
9                       end;
10
11                 end;
12
13       r 1839  1.250  5+43
14
15*      pl1 blowup -table
16       PL/I, Version 2
17
18       WARNING 307
19       The variable "a" has been referenced but has never been set.
20       r 1840  10.351  5+355
21
22*      blowup
23
24       Error:  out_bounds_err by blowup|16
25       referencing stack_4|777777 (in process dir)
26       r 1840  1.087  3+35
27
28*      debug
20*      /blowup/16&t,s
30                 j = a(loop_index);
31*      loop_index
32       1413    113          -769
33       .q
34       r 1841  .840  4.277  120
```

Figure 4.3: A simple example of source language debugging.

that subscript checking should be done by PL/I, the compiled code will attempt to do something with the negative subscripts, in this case scanning downwards in the stack until the bottom is reached; a hardware trap will then catch the errant program.

Note that, in preparation for debugging a new program, the "table" option of the compiler is used, on line 15. This option requests the compiler to leave its symbol table embedded in the program, for run-time use. A warning of trouble is provided by the compiler on line 19, but this does not deter us from trying the program, on line 22. As predicted, an out-of-bounds fault occurs when referring to the next location in the stack after location zero. A standard Multics notation for memory locations is exhibited twice in the error message, once on line 24 and again on line 25. On line 24 we see the string:

```
blowup|16
```

which is interpreted as "in the segment named blowup, at offset 16 (octal) locations from the base". (This notation should be read "blowup offset 16".) Thus line 24 gives us the address of the offending instruction, while line 25 tells us the out-of-bounds address which it attempted to reference.*

---

* The message on lines 24 and 25 is printed by the Multics "default error handler" which means that the program which was running had not explicitly arranged to respond to the particular error which occurred. (A PL/I "on condition" statement is used for explicitly catching such errors.) The following errors are commonly encountered:

out_bounds_err	an out of range subscript or uninitialized subscript or pointer variable was probably used, leading to a reference to a legal segment number but an illegal word address within the segment.
linkage_error	a call occurred to a subroutine which could not be found. It is possible to type "hold", write the missing subroutine, compile it, and then restart the program which got the linkage error.
record_quota_overflow	The user's secondary storage allocation has been exceeded. If one types "hold", he may then list his directory, delete something, and then restart the program which ran into the overflow.

For the cause of and recovery from other errors, the MPM sections on handling of unusual occurrences and condition names should be consulted.

To find out what has gone wrong, we now use the debug command on line 28: there is no reply when the command name is typed, so the next line, 29, contains the first request to debug. The syntax of debug requests is straightforward, though cryptic at first. One specifies first a Multics memory address, then what to do at that address. On line 29, the string "/blowup/16&t" specifies the address: starting from segment named "blowup", go to the 16th location in the text. The string ",s" after that address specifies that the contents of that location should be printed out, in symbolic (source-instruction) format. Thus we see, on line 30, the line of code which caused the out-of-bounds fault to occur.

To inspect individual variables to see what has gone wrong, one merely mentions them by name, as on line 31, and debug will print out their position (1413 locations from the base of the stack, 113 from the current stack frame base) and value (-769 in the example.) Note that this request follows the general form of all debug addressing requests, but that defaults are used profusely. In the absence of a segment name, the last one mentioned (/blowup/) is used; in the absence of specific instructions for output format, a format appropriate to the variable (decimal integer) is used; in the absence of any other instruction, output printing is assumed. In the place where the variable name is typed, an arbitrarily complex identifier may be used. Thus, if the program contained a based, two-dimensional array named x, one could look at an element of that array by typing:

```
p->x(i, j)
```

The debug command would look up each variable in turn, evaluate the subscripts, then fetch the array element in question, using the current value of "p" as a base.

Finally, having satisfied ourselves as to the status of the program, we exit debug by typing the request on line 33. All debug requests not related to memory locations are preceded with a period. Since we did not type hold following the error, the command language interpreter will release the stack contents upon return from debug. We have no further use for the errant program, and for this example it makes no sense to repair it and continue, so a stack release is the appropriate action.

As an example of breakpoint debugging, consider the pair of programs in figure 4-4. According to plan, one calls the program "trev" with a string of words; trev calls recursive procedure "rev" to reverse the order of words in the string; then it prints the reversed string. When we try to run the program, we obtain the particularly discouraging comment on line 29 -- apparently the recursive procedure has run wild, and run out of stack space. A new process, with a new stack, is created automatically but unfortunately the current version of Multics discards the old process and its stack, which contain most of the clues needed to

```
 1* print trev.pl1
 2
 3  trev:     procedure(string);
 4
 5  declare   string character(*) unaligned,
 6            rev entry(character(*)) returns(character(32) varying);
 7            put skip list(rev(string));
 8            put skip;
 9
10                   end;
11
12  r 1819  1.732  4.670  106
13
14* print rev.pl1
15
16  rev:      procedure(string) returns(character(32) varying);
17
18  declare   string character(*);
19            i = index(string," ");
20            if i = 0 then return(string);
21            else return(rev(substr(string,i))||" "||
                                (substr(string,1,i)));
22
23                   end;
24
25  r 1820  .513  4.040  133
26
27* trev "now is the time"
28
29  Fatal error.  Process has terminated.  Out of bounds fault on stack.
30  New process created.
31  r 1820  2.006  5.263  127
32
33* debug
34* /rev/&a5<
35  Break 0 of rev set at 34 from  34   600100236100   ldq    sp|100
36* ..trev "now is the time"
37  Break 0 at line 5 of rev, 220|34
38* string
39    3561  -447 "now is the time"
40* .c
41  Break 0 at line 5 of rev, 220|34
42* string
43    4372  -6 " is the time"
44* .be string;.c
45* .c
46  Break - at line 5 of rev, 220|34
47  string;.c
48    4542   -6 " is the time"
49  Break - at line 5 of rev, 220|34
50  string;.c
51    4112   -6 " is the time"
52  QUIT
53  r 1822  13.873  41.426  557
```

Figure 4-4: Breakpoint debugging

debug the program. (Future versions of Multics will save some information about the defunct process.)

Since there is no clue as to why the recursive procedure is not properly stopping its recursion, we enter debug and, on line 34, place a breakpoint in procedure rev at program line 5. (The string "&a5" means line 5, the character "<" means set a break.) Debug responds by printing the old contents of the location it had to modify; this information is not of interest to us. Now, we call, from inside debug, out to procedure "trev", on line 36. (Any Multics command or program may be called from within debug by typing the two periods at the beginning of the request line.)

Now, debug calls to trev, and the next thing we know, the break point is reached, putting us back into debug, which prints the message on line 37. We look at variable "string" to see what has been handed to the subroutine as an argument. Since the string printed on line 37 is exactly what we expected, we type .c on line 40, meaning "continue the program until the break point is reached again." Again the break point is encountered, and the string inspected, and it looks OK. Being impatient, we now type the special "macro" request on line 44: "whenever a break occurs, print the contents of "string", then continue." We again start the program on its way, and its faulty behavior immediately becomes apparent as the debugger prints lines 46-51: the argument string is not changing after the second iteration. Inspection of the program reveals the trouble; the blank character should have been stripped from the front of "string" before recursively calling; changing the second argument of the first substr in line 21 to i + 1 will fix the program.

On line 52, we have exited from our looping program by quitting out of it. This leaves us at a higher stack level, with both our program and our invocation of the debug command somewhere earlier in the stack. It also leaves program rev with a breakpoint inserted in its code. To be careful, we should now type the program_interrupt command, which will return us to the most recent invocation of debug, so that we may reset the breakpoint gracefully. Failure to reset the breakpoint would lead to mysterious difficulties ("mme2" faults) if we later ran the program without using debug to control it. Of course we can also recompile the program, in which case we also get a new copy without breakpoints. Figure 4-5 continues the example of figure 4-4, using the program_interrupt command to return to the debugger, on line 55. Now, to see what the stack looks like, we request debug to trace the stack contents, with the .t request on line 56. Lines 60-78 are the successive entries currently on the stack with the oldest entry first. The first four entries, on lines 60-63, represent the procedures provided by the Multics system to set up the standard command environment, and are unimportant to us right now, except to notice that line 63 is the command language interpreter. On line 64 is the debug command, the result of typing "debug" back on line 33. While in debug, we called out, on line 36, to the

```
54
55* program_interrupt
56* .t
57
58  Depth Segno  Offset    Name
59
60     0   200     120 real_init_admin_|15771
61     1   200     260 process_overseer_|15057
62     2   200     460 listen_|2304
63     3   200     760 command_processor_|3127
64     4   216    1300 debug|6651
65     5   200    2630 command_processor_|3225
66     6   231    3150 full_command_processor_|3006
67     7   231    3600 bound_full_cp_|2366
68    10   232    4010 trev|117
69    11   220    4230 rev|115
70    12   220    4400 rev|115
71    13   220    4550 rev|115
72    14   220    4720 rev|115
73    15   220    5070 rev|115
74    16   220    5240 rev|115
75    17   220    5410 rev|115
76    20   220    5560 rev|115
77    21   220    5730 rev|115
78    22   220    6100 rev|34
79*   .q
80    r 1825  2.438  7.611  257
```

Figure 4-5: Tracing the call stack.

program we were debugging. The debug command called out to the standard command language interpreter, since line 36 contained a standard Multics command line. Thus, line 65 describes a second generation of the same program we saw earlier on line 63. Note, however, that the location in the stack (the column labeled Offset) is different for the two generations of the command language interpreter: the two generations will therefore use different copies of automatic variables.

The command line typed on line 36 provides as a single argument a string (including blanks) enclosed in quotation marks. The command language interpreter is organized in several modules, such that for the most common (and simplest) syntax, only a small part of the interpreter is needed. Whenever a more elaborate syntactical structure is encountered, a more elaborate section of the interpreter is invoked. In the case at hand, the quoted string argument triggers a need for the more elaborate interpreter, so on line 66 we see that a program named full_command_processor_ was called, and it entered an internal block which debug has tagged with the name bound_full_cp_.

Finally, the command language interpreter constructed a call to trev, the program being debugged, on line 68. Program trev then called rev, which called itself recursively several times before we hit the quit button. Notice than the number of recursive calls to rev found in the stack (10 in this example) is greater than the number of times that debug breakpoints were encountered on lines 35-49. Recall that on line 44, debug was instructed to let the program run without stopping at breakpoints, except for printing the contents of the variable named string. The Multics typewriter output package operates asynchronously, which means that it begins typing an output message, and simultaneously returns control to the process originating the message. The process can then go on to its next step, perhaps producing more messages, which the typewriter package collects in a queue for the typewriter. Thus in our example, the program had gotten well ahead of the typewriter when both it and the typewriter output were stopped.

An alternative way of examining the contents of the stack is to use the command trace_stack, which provides a wealth of information about each stack level: the arguments used in the call from the last level, the symbolic instruction which caused the call, a list of enabled on-conditions at the stack level, details of any faults or signals which occurred, etc. The MPM write-up of trace_stack provides more details. The trace_stack command is especially useful for situations where something mysterious has happened, which requires help from an expert who is not available at the moment. The output from trace_stack is often sufficient to diagnose, or provide clues in the diagnosis of very complicated problems.

The reader should not feel that these two short examples have completely explained the ins and outs of using the debug command. However, until he has had time to more thoroughly review the MPM write-up of debug, he may find the samples useful to imitate while debugging his own programs.

One final comment about symbol tables is of significance: the symbol table (created by the "table" option of PL/I) is stored in the end of the program, in an otherwise unused area. If it is not explicitly used, as by the debugger, then it will not cause any extra paging activity. It will, however use up secondary storage space. Thus, it is recommended that while a new set of programs is being debugged, the table option be used in all compilations. After one is reasonably satisfied that all of his programs are working properly, he may wish to recompile without the table option, to save long term secondary storage charges.

The reader should also refer to the MPM Reference Guide section on the Multics Command Repertoire, where a list of other useful debugging tools is provided.

Absentee Use of Multics

A common programming pattern is to develop a program on-line, using debugging tools and the ability to interactively try a variety of test cases to check on a program's correctness. After the program is working, one may wish to do a large "production" run. Since the production run may produce much output or take much time, the programmer does not wish to wait at his terminal for the results. For such cases, he may develop an absentee job, and submit it for execution. This technique has several implications:

- The job is not under control of a terminal, so an absentee job control segment must be constructed.

- Since there is no terminal available, all input and output must come from and go to the storage system.

- The absentee job is placed in a queue and run as background to the normal interactive work of the system. This technique provides a buffer of pre-emptable resources for interactive peak loads, and meanwhile helps keep the system fully utilized. For these reasons, the charging rate for absentee jobs is normally substantially lower than for interactive work.

The job control language of the Multics absentee facility is identical to the command language typed at the console. In general, an absentee job is given a name, say "a". When run, an ordinary Multics process is logged in, but its input stream is attached to a segment named a.absin, and its output stream to a segment named a.absout. Thus to control an absentee job, one must first create the absentee input segment which contains the commands to be executed.

In figure 4-6 is a version of the primetest program used before. It has been modified to be a "production" program by adding a do loop. One might interactively start this program to check that it is producing the expected results:

```
primetest

          1         is a prime.
          2         is a prime.
          3         is a prime.
          4         is not a prime.
          5         is a prime.
          6         is not a prime.
          7         is a pri
QUIT
r 1519  5.834  20.147  1061
```

To submit the job for absentee execution, the user first constructs a control segment to be used for input to the job. The only input in this case is the command line required to

```
primetest:          procedure;

declare   prime_input fixed binary;
declare   (sqrt,mod) builtin;
declare   (sysprint) file;

          do prime_input = 1 to 150;
               if prime(prime_input)
                    then put list (prime_input, "is a prime.");
                    else put list (prime_input, "is not a prime.");
               put skip;
          end;
          return;

prime:    procedure(trial_prime) returns (bit(1));

declare   trial_prime fixed binary,
          trial_factor fixed binary,
          last_factor fixed binary;

          last_factor = sqrt(trial_prime);
          do trial_factor = 2 to last_factor;
               if mod(trial_prime, trial_factor) = 0
                    then return ("0"b);
               end;
          return ("1"b);

          end prime;

          end primetest;
```

Figure 4-6: Production version of the primetest program.

execute program primetest. Thus, he creates a segment named prime.absin, using an editor:

```
 1*        edm prime.absin
 2         Segment not found.
 3         Input.
 4*        primetest
 5*        logout
 6*        .
 7         Edit.
 8*        w
 9*        q
10         r 1537  2.373  27+214
11
12*        enter_abs_request prime.absin
13         23 already requested.
14         r  1538  4.841  9.083  319
```

And now, he may go about his business, whether working at his terminal or logging out, as he chooses. Some time later, after the jobs ahead of his are processed, a new process will be logged in and his two commands will be executed. When the job is finished, a segment named prime.absout will appear in his directory, which he may print on his terminal, or send to the high-speed printer, as desired.

Our example absentee job uses only the most rudimentary features of the absentee facility. One can also supply arguments to be substituted inside the absentee control segment, make absentee job steps conditional, delay absentee work until a chosen time, and develop a periodic absentee job which is run, say, once every two days.

Sometimes, a very elaborate absentee control segment is constructed, and the user may wish to verify that his absentee job will operate properly. One useful technique for checking out an absentee control segment is to use it as a control segment for the exec_com command, a macro_command facility which accepts the same kind of control segment as does the absentee facility. The MPM Reference Guide sections on enter_absentee_request and exec_com contain further information on these facilities.

Dynamic Linking and Binding

A particularly potent programming tool of Multics is the dynamic linking facility. Dynamic linking consists of delaying the search for and mapping of a subroutine (or data segment) until the first call for that subroutine (or use of that data segment) occurs. Dynamic linking is accomplished by having the compiler leave in the object code of a compiled program a special bit pattern which, if used in an indirect address reference, causes a machine fault (trap) to the dynamic linker. The linker inspects the location causing the fault, and from pointers found there, locates the symbolic name of the program being called or the data segment being referenced. It then locates the appropriate segment, maps it into the current address space, and replaces the indirect word with a new one containing the address of the program or data entry point, so that future references will not cause a dynamic linking fault.

There are many ways in which dynamic linking can be used, but the following three are probably most significant:

. to permit initial debugging of collections of programs before the entire collection is completely coded.

. to permit a program to include a conditional call to an elaborate error handling or other special-case handling program, without invoking a search for or mapping of that program unless the condition arises in which it is actually needed.

. to permit a group of programmers to work on a collection of related programs, such that each one obtains the latest copy of each subroutine as soon as it becomes available.

Whenever related subprograms are separately translated, they are normally linked by the Multics dynamic linker at the time they are executed. If a set of related programs is known to always require certain links, then a program known as the binder may be used to pack them into a single segment, permanently link any cross references, and condense any common outward references into a single outbound link. In return for the loss of flexibility which comes with such permanent binding, one reduces both the space required for the programs and the number of library searches which must be undertaken to run the collection of programs. In addition, binding of separately translated subroutines retains most of the advantages of separate translation. (An alternative scheme would be to collect the procedures together into a single giant procedure, and then recompile. This alternate scheme has the disadvantage that a very long recompilation is needed for every one-line change to any part of the collection of programs.)

To provide a brief example of the meaning of dynamic linking, consider the sample console session of figure 4-7. Procedure k, on lines 9-14, reads an integer from the console, and then calls one of three different subroutines. Only one of these subroutines, named y, actually has been written. On line 30, k is invoked, it asks for input, and the input value which causes y to be called is typed on line 31. Line 32 provides evidence that y was called. Note that, although the statement on line 11 was executed, the conditional test failed, and a call to procedure x (which has not yet been written) did not occur. Since linking is done on demand, and no demand for x occurred, the fact of its non-existence has not kept us from running our procedure y.

```
 1*       print k.pl1
 2
 3        k:          procedure;
 4
 5        declare     (x,y,z)  external entry;
 6        declare     i fixed binary;
 7        declare     (sysprint,sysin)      file;
 8
 9                    put list ("What now? ");
10                    get list (i);
11                    if i = 1 then call x;
12                    if i = 2 then call y;
13                    if i = 3 then call z;
14                    return;
15
16                    end k;
17
18        r 927  1.075  3.994  178
19
20*       print y.pl1
21
22        y:          procedure;
23                    declare  sysprint  file;
24                    put list ("y has been called.");
25                    put skip;
26                    end y;
27
28        r 927  .699  1.806  79
29
30*       k
31(*)     What now?  2
32                    y has been called.
33        r 928  .858  2.012 112
34
35*       k
36(*)     What now?  3
37
38        Error: Linkage error by k$k|165
39        Referencing z|z.
40        Segment not found.
41        r 928  1.318  5.855  252
42
43*       hold
44        r 928  .199  2.062  38
45
```

Figure 4-7: Dynamic linking example.

```
46*       edm z.pl1
47        Segment not found.
48        Input.
49*       z:        procedure;
50*                 declare sysprint file;
51*                 put list ("Z has been called");
52*                 put skip;
53*                 end z;
54*       .
55        Edit.
56*       w
57*       q
58        r 929  1.280  5.274  223
59
60*       pl1 z
61        PL/I, Version 2
62        r 930  7.036  20.651  263
63
64*       start
65                  Z has been called
66        r 931  .875  2.132  150
```

Figure 4-7, Continued.

On line 35, k is invoked again, this time with a request to call procedure z. Since z does not yet exist, the default error message on lines 38, 39, and 40 explains that a linkage error occurred, when subroutine k attempted to reference subroutine z. Note, by the way, that line 38 uses one convention, k$k, to refer to segment k, entry point k, while line 39 uses a different convention, z|z, to refer to segment z, entry point z. These two conventions should be considered equivalent. (One arose from a standard compiler syntax, while the other arose from a standard assembler syntax.)

To illustrate that a linkage error is normally recoverable, a hold command is typed on line 43, and then a program named z is typed in and compiled on lines 46-62. (See figure 4-7, continued.) When start is typed on line 64, we see that the original call (from line 14 in procedure k) to subroutine z has now succeeded.

For more information on the details of dynamic linking and binding see the MPM Reference Guide sections on object segments, system libraries and search rules, and the command bind.

A Simple Text Editor

Our next sample program is a text editor similar to, but simpler than, the edm command used in Chapter Three. It is a typical example of an interactive program which makes use of the Multics storage system via the virtual memory. In overview, the editor creates two temporary storage areas, each large enough to hold the entire text segment being edited. It copies the segment into one of these areas, so as not to harm the original and then, as the user supplies successive editing requests, constructs in the other area an edited version of the segment. When the user finishes a pass through the segment, the editor interchanges the roles of the two storage areas for the next editing pass. When finished the appropriate temporary storage area is then copied back over the original segment.

For this example, we have available a program listing as produced by the PL/I compiler. The program itself is derived from the edm command of Multics, and it exhibits several different styles of coding and commenting, since it has had many different maintainers.

The reader will also notice that some comments appear to be critical of the program style or of interfaces to the Multics supervisor. These comments should be taken in the spirit of illumination of the mechanisms involved. Often they refer to points which could easily be repaired, but which have not been in order to provide a more interesting illustration. Most of the points criticized are minor in impact. Finally, some comments mention effectiveness of compiled code for certain constructs. Experience has shown that as PL/I compiler technology advances, the range of constructs which produce efficient compiled code increases. Such comments, then, should be considered to be dated, and subject to change.

The program begins on page 40 following the comments.

Line number	
first unnumbered line	The compiler both records here and encodes into the binary object program the date and time of compilation and the version of the compiler used. The print_link_info command may be used to print the date and time of compilation stored in the object program. If it is not identical to that printed at the top of the listing, then the listing is for a different compilation, and should be suspected as being possibly a different program.
fourth unnumbered line	The command "pl1 eds -map -optimize" was typed at the console. This line records the fact that the map and optimize options were used. The map option caused a listing and variable storage map to be produced. A source segment named eds.pl1 was used as input; the compiler constructed output segments named eds.list (containing the listing) and eds (containing the compiled binary program.)
1	No explicit arguments are declared here, even though eds should be called with one argument. The argument

is instead picked up with a library subroutine which can return an error indication if the argument is missing. Since eds is used as a command, it is a good human engineering practice to check explicitly for missing arguments; the PL/I language has no feature to accomplish this check gracefully. (See lines 84-89.)

4 To avoid errors when program maintenance is performed by someone other than the original coder, all variables are explicitly declared. This practice not only avoids surprises, but also gives an opportunity for a comment to indicate how each variable is used.

6 One default which is used here (and is subject to some debate) is that the precision of fixed binary integers is not specified, leading to use of fixed binary(17). This practice has grown up in an attempt to allow the compiler to choose a hardware supported precision, and in fear that an exact precision specification might cause generated code to check and enforce the specified precision at (presumably) great cost. In fact, most such considerations are not relevant to the Multics implementation; for all aligned variables with precisions less than one word (fixed binary(35)), the compiler generates code which uses word length hardware and does not enforce the precision specification. Ideally, one should consider the expected range of each variable and specify an appropriate precision for it, rather than depending on a forgiving implementation which accidentally supplies more precision than requested.

7 Most character strings in this program are declared aligned so as to insure that the fastest possible accessing code will be produced. The only exceptions are character strings which are to be used as arguments to supervisor entries which require unaligned strings. (See lines 25, 62, and 440). In programs such as this one, the storage space loss due to use of the aligned attribute on a few character variables is generally trivial compared with the space required to hold accessing code and time required to execute it. Obviously this comment might not hold in a case where many hundreds or thousands of character strings are involved.

9 All line buffers are designed to hold one long typed line (132 characters for input terminals with the widest lines) plus a moderate number of backspace/overstrike characters. To support memorandum typing, the buffers permit a 70-character line which is completely underlined. Note also that the current typewriter input conversion package has a defect which requires that the original input line, before erase and

kill editing, and before overstrike canonicalization, fit into the character buffers provided by the user for correct conversion to take place.

10 The variable named code has precision 35 bits, since it is used as an output argument for several supervisor entries which return a fixed binary(35) variable. It would seem appropriate, on a 36-bit machine, to use fixed binary(35) declarations everywhere. However, use of fixed binary(35) variables for routine arithmetic should be avoided since, for example, addition of two such variables results in a fixed binary(36) result, forcing the compiler to generate code for double precision operations from that point on. One must be careful of the PL/I language rule which requires the compiler to maintain full implicit precision on intermediate results.

12 Automatic variables with initial values are set to their initial values every time the program is entered. This method is at least as effective as a series of initialization statements at the beginning of the program, and perhaps clearer to the reader.

17,18 All editing is done by direct reference to virtual memory locations. The variable from_ptr is set to point to a source of text, and the based variable from_seg is used for all reference to that text.

18 The general operation of the editor is copy the text from one storage area to another, editing on the way. The names from_seg and to_seg are used for the two storage areas.

25 It is necessary for this program to know the I/O stream name on which input will be typed. Programs which perform less sophisticated input operations can often get along with system supplied defaults for the I/O stream names. (See comment on line 440.)

34 The PL/I language provides no direct way to express literal control characters. The technique used here, while adding clutter to the program listing at least works and is machine independent.

36 One set of supervisor interfaces calls for 24 bit integers; this declaration guarantees that no precision conversion is necessary when calling these interfaces. (See line 97).

40 Supervisor entries generally use fixed, rather than varying, strings. (In an earlier compiler implementation, varying strings were very inefficient, and based varying strings were forbidden.) Thus, when

calling older supervisor entries it is occasionally necessary to simulate a varying string by using a fixed string and an integer count of the number of characters in the string. (See lines 84 and 93 for the single example in this program.)

51 Subroutines com_err_ and ioa_ are called with a different number of arguments each time, a feature not normally permitted in PL/I. The Multics implementation, however, has a feature to permit such calls to be compiled. The "options" clause warns the compiler that the feature is to be used for this external subroutine.

52 All subroutines other than com_err_ and ioa_ are completely declared in order to guarantee that the compiler can check that arguments being passed agree in attribute with those expected by the subroutine. Warning diagnostics are printed if the compiler finds argument conversions to be necessary.

52 The procedure cu_ (short for command utility) has several different entry points. The Multics PL/I compiler specially handles names of external objects which contain the dollar sign character. The dollar sign is taken to be a separator between a segment name and an offset name in the compiled external linkage. Thus, this line declares the entry point name arg_ptr in the segment name cu_.

53 For many procedures, the segment name and entry point name are identical, so the compiler also permits the briefer form cv_dec_, which is handled identically to cv_dec_$cv_dec_.

55 The hardcore (ring zero) supervisor entries are all easily identifiable since they are entered through a single interface segment named hcs_. Segment hcs_ consists of just a set of transfers on to the subroutine wanted. A transfer vector is used to isolate, in one easily available location, all gates to the Multics supervisor. Also, it is in principle possible to dynamically replace a supervisor routine, by changing a single transfer instruction.

56 Note that supervisor entry hcs_$make_seg takes unaligned character strings for its first three arguments. This property will turn out to be a nuisance later (line 95) since the library subroutine which constructs the arguments for hcs_$make_seg returns aligned character strings. See the comments on lines 93 and 95 for more information.

67 This implementation-dependent declaration is a based structure, designed to overlay on top of a 64K Multics segment, and thereby allow construction of a pointer to the midpoint of the segment. The declaration depends on fixed binary variables of precision less than 36 bits occupying one word each.

68 The comment on this line consists of a single ASCII control character, for form feed (octal 014). The closing syntax for the comment appears at the top left edge of the next page. Such "vertical punctuation" between major parts of a program is recommended for program readability.

73,74 The segment name is copied into an intermediate storage space since it may be used in an error comment. Note that we should not use the variable ename as the second argument in the call to hcs_$make_seg, since ename is aligned and hcs_$make_seg requires unaligned input arguments.

74 The first step in the program is to obtain a pointer to a "scratch" or temporary segment in which intermediate copies of the text being edited may be stored. Subroutine hcs_$make_seg will create a segment, if one does not already exist with the specified name. The binary string specifies that if a segment is created, the system should permit read and write access to the segment. The system creates the segment, maps it into the address space of this process, and returns a pointer in the variable from_ptr. The first argument to hcs_$make_seg specifies the name of the directory in which the segment should be located. A null string, as in this case, indicates that the segment is to be created in the process directory, a suitable home for temporary segments. The third argument is a place for a reference name, which would be specified if there were to be later references to the segment to be accomplished by dynamic linking. Since no such reference will occur, a null string is specified.

74 Although our program has no declared static variables, the segment eds_temp is now effectively a program-created static variable. If, for example, one were to quit out of the editor, issue a "hold" command to maintain the stack level, and then reinvoke the editor at a new, deeper, stack level, the second invocation of the editor would, upon encountering line 74, obtain a pointer to the same segment, eds_temp, that is being used by the earlier, interrupted invocation. If the second invocation of eds overwrites eds_temp, then upon later return to the earlier, interrupted invocation one would probably be in deep trouble. Three different techniques could have been

used to avoid this trouble: 1) document the restriction that the editor cannot be used recursively, or 2) put a check in the editor to see if a previously created eds_temp exists, and give warning if one does, or 3) implement an automatic, rather than a static, temporary segment, by using a guaranteed unique name (Multics subroutine unique_chars_ can be useful here) for the temporary segment.

75 If there was trouble creating a buffer segment, hcs_$make_seg returns a null pointer. It also returns a status code, but since a non-zero status code is returned in some non-error cases (e.g., when a segment named eds_temp was already there) the easiest test for a disastrous error is on the returned pointer.

77 The subroutine com_err_ should be called to print out the error message associated with the returned status code. However, the calling sequence is quite long, so an internal subroutine, called from many places in eds, minimizes the amount of generated call setup code.

78 One exits from a Multics command by simply returning to its caller. (See also line 351).

80 (See comment re line 67). Here, in an economy move, we create a pointer to the midpoint of the segment just created. We thus avoid the need to create two temporary segments for editing. At this point from_seg points to the base of the segment and to_seg points to the midpoint. The two halves of the segment will be used as two buffers for editing. Note that this strategy restricts the maximum size of a segment which may be edited, yet the editor nowhere checks to see if this maximum size is being exceeded, an unfortunate omission. Since lack of a check could cause overwriting of data, a program with this defect would not be considered acceptable for the Multics command library.

84 When a user types a command such as "eds alpha" the first string of characters is taken as the name of a procedure to be called, while succeeding strings are taken as character string arguments to that procedure. Rather than declaring eds to have one argument, which would not permit a graceful exit if no argument were typed, we pick up the argument with subroutine cu_$arg_ptr, which returns a pointer to the beginning of the unaligned character string representation of the first argument, which eds considers to be the name of the segment to be edited.

85 For many subroutines, any non-zero status code indicates that the subroutine could not properly

complete, and recovery action is appropriate. In this case, the most likely error is that the argument is missing.

88 When an error occurs now, we do not immediately return, since we have created a temporary segment, and should clean up after ourselves first. Thus the transfer to quit1 rather than a return. (See line 348.)

93 Assuming that a pointer to an argument was returned, we must now convert that argument to a standard (directory name, entry name) pair. The subroutine expand_path_ implements the system-wide standard practice of interpreting the typed argument as either a path name relative to the current working directory, or an absolute path name from the root, as appropriate.

93 The third and fourth arguments to expand_path_ are (unnecessarily) required to be pointers to the character strings in question, rather than the strings themselves. Because pointers are the formal arguments, neither the reader, nor a mechanical argument checking program, can detect whether or not the real arguments being passed behind the pointers match in type with those expected by the writer of expand_path_. Examination of the MPM write-up for expand_path_ tells us that aligned character strings are required for the third and fourth arguments, and an unaligned character string for the first one. (This interface is a left-over from a time when character string arguments were very expensive to pass directly.) In such cases, it is a good practice to represent the arguments as shown, for clarity, rather than by setting and passing pointer variables whose purpose is not clear to the next maintainer of the program. In general, it is a good practice to consider pointer variables to be escapes around missing language or system features, and therefore to isolate their use in a way which makes clear what is being escaped around. This program follows this practice whenever possible, but some older supervisor interfaces force a departure.

95 We now call hcs_$make_seg again, to either create or get a pointer to the source segment to be edited, this time specifying the directory and entry names returned by expand_path_. As mentioned earlier, hcs_$make_seg requires unaligned character strings in its first three arguments, but ename and buffer are the aligned return values from expand_path_. Therefore, the compiler, noting that the declaration on line 56 disagrees with those on lines 9 and 15, will automatically generate code to copy the aligned strings over into unaligned temporary variables for the duration of the call. The compiler will normally print a warning diagnostic when

it generates such code, in case the programmer doesn't realize that he is forcing a type conversion. To suppress the warning message, the first two arguments to hcs_$make_seg have been placed in parentheses, which are taken by the compiler to be an explicit request for conversion; therefore no message is printed.

Occasionally one will encounter an extremely bad practice which has been used to get around the argument copying: subroutine hcs_$make_seg may be misdeclared to take aligned arguments. Since it happens that the Multics implementation of aligned character strings is identical to unaligned character strings which start on a word boundary, the misdeclaration happens to work. This mapping together of aligned and a subset of unaligned does not necessarily hold in other PL/I implementations, and it does not hold in Multics for variables other than strings. In any case, use of such constructs is an outstanding example of bad programming practice for two reasons: first, it relies on obscure properties of the local implementation; second, one would like to have available a mechanical technique for detecting accidentally mismatched arguments; intentionally mismatched ones would then frustrate mechanical verification.

97 The storage system provides for every segment a variable named the bit count. For a text segment, by convention, the bit count contains the number of information bits currently stored in the segment. Subroutine hcs_$status_mins obtains the value of the bit count.

97 Clearly, the calls to expand_path_, hcs_$make_seg, and hcs_$status_mins could have been a single subroutine call to a subroutine which performs all three functions. Such an interface would eliminate the need for this procedure to care about (and provide storage for) such things as the number of characters in the typed argument string, and the name of the directory containing the segment being edited. The hassle about aligned and unaligned strings could be avoided, too.

99 If the segment to be edited did not previously exist, (that is, the call to hcs_$make_seg created the segment rather than merely returning a pointer to it) then the bit count will be zero, and the editor assumes that is should start in input mode.

103 This statement converts the bit count to a character count. Note that we have here embedded knowledge of the number of hardware bits per character in this program. If the system-wide standard had been to store a character count with a segment instead, it would not

have been necessary to have an implementation-dependent statement here. Unfortunately, a stored character count would get the system into the business of maintaining an interpretation of the segment's contents, which it currently does not do. A still better strategy would have been to store a character count in the segment itself, say in the first word, thus maintaining the view that a segment maintains its own interpretation.

103 The PL/I language specifies that the result of a divide operation using the division sign is to be a scaled fixed point number. To get integer division, the divide built-in function is used instead.

104 Here, we invoke some of the most powerful features of the Multics virtual memory. This simple assignment statement copies the entire source segment to be edited into the temporary buffer named from_seg. Highly optimized machine code performs the actual copy loop. Note that we are regarding the entire text segment as a simple character string of length csize. We may regard it this way because the storage representation for permanent text segments is chosen to be identical to that of a PL/I fixed character string.

106 Be sure to read the comments embedded in the program, too.

109 Subroutine ioa_ is a handy library output package. It provides a format facility similar to PL/I and FORTRAN format statements, and it automatically writes onto the I/O stream named user_output, which is normally attached to the interactive user's terminal. When used as shown, it appends a new line character to the end of the string given. Programmers who are more concerned about speed than about compatibility with other operating systems use ioa_ in preference to PL/I "put" statements, because ioa_ is a less general facility which does not touch nearly as many distinct storage pages.

111 Here we have another interface which (unnecessarily) requires use of a pointer in its first argument. Again, one result of this obsolete practice is that complete type-checking by the compiler is not possible for that argument. Some of the more sophisticated I/O system entries use a pointer in the same position, but with a better reason: those entries can transmit variables of various types on different calls, so no single variable declaration could suffice.

111 Subroutine ios_$read_ptr is often used for input rather than the PL/I statement "read file (sysin) into ..."

again because the ios_ entry has fewer options and therefore touches fewer storage pages. The PL/I facility ultimately calls on the Multics ios_ package anyway. (Again, if one wished to write a program which would also work on other PL/I systems, he would be better advised to use the PL/I I/O statements instead.)

112 For human engineering, blank lines are ignored by the editor. Since complete input lines from the typewriter end with a new line character, the length of a blank line is one, not zero.

114 The code to isolate a string of characters on the typed input line is needed in four places, so an internal subroutine is used. This subroutine is not recursive, which makes it possible for the compiler to construct a one-instruction calling sequence to the internal procedure. Certain constructs (e.g., variables of adjustable size declared within the subroutine) will force a more complex calling sequence. For details, one should review the documentation on the Multics PL/I implementation.

116 Although the dispatching technique used here appears costly, it is really compiled into very quick and effective code -- 4 machine instructions for each line of PL/I. For such a short dispatching table, there is really no point in developing anything more elaborate. If the table were larger, one might use subscripted label constants for greater dispatching speed.

121 Human engineering: the typist is forced to type out the full name of the one "powerful" editing request which, if typed by mistake, could cause overwriting of the original segment before that overwriting was intended.

131 The format and decimal conversion facilities of ioa_ are used in a simple way in this example. The "not" sign in the format string indicates where a converted variable is to be inserted; the character following the not sign indicates the form (in this case, a character string) to which the variable should be converted. The first argument is the format string, remaining arguments are variables to be converted and inserted in the output line:

132 Whenever a message is typed which the typist is probably not expecting, it is good practice to discard any type-ahead, so that he may examine the error message, and redo the typed lines in the light of this new information.

138 The general strategy of the editor is as follows: lines from the typewriter go into the variable named "buffer" until they can be examined. Another buffer, named "line" holds the current line being "pointed at" by the eds conceptual pointer. Subroutine "put" copies the current line onto the end of to_seg, while subroutine "get" copies the next line in from_seg into the current line buffer.

142,143 If ios_$read_ptr returned a varying string rather than a fixed string and a count, these two statements could reduce to "line = buffer". More use of varying or adjustable strings would probably simplify the appearance of this program quite a bit.

150 The procedure get_num sets up the variable n to contain the value of the next typed integer on the request line. Such side-effect communication is not an especially good programming practice.

152 The delete request is accomplished by reading lines from from_seg, but failing to copy them into to_seg. If deletion were a common operation, it might be worthwhile to use more complex code to directly push ahead the pointer in from_seg, and thus avoid a wasted copy operation.

161 More side-effect communication: the variable edct is always pointing at the last character so far examined in the typed request line.

177,187 All movement of parts of the material being edited is accomplished by a simple string substitution, using appropriate indexes.

206 The locate request is accomplished by use of the index built-in function, used on whatever is still unedited in from_seg.

319 A negative number in the "next" request results in moving the conceptual pointer backwards. The resulting code is quite complex for two reasons:

a) The eds editing strategy requires interchanging the input and output segments before scanning backwards, so that the backward scan is with regard to the latest edited version of the segment.

b) At the time this program was written, there was no PL/I feature to perform an "index" function starting from the end of a character string rather than the beginning. The "reverse" built-in function could now be used.

348 Before exiting from the editor, the temporary segment should be cleaned up. The question of whether the temporary segment should be deleted or merely truncated is a slightly fuzzy one. Since the editor is almost certain to be used several times in a process, the choice was made here to not delete it, so that later invocations of the editor will result in a faster response from make_seg. If, on line 74, we had used a unique name for the temporary segment, then we should surely delete it here, since no one will ever ask for a segment by that name again.

362 Another human engineering point: since the user may have typed several lines ahead, the error message includes the offending request, so that he can tell which one ran into trouble and where to start retyping.

363 Note a small "window" in this sequence of code. If the editor is delayed (by "time-sharing") between lines 362 and 363, it is possible that the message on line 362 will be completed, and the user will have responded by typing one or more revised input lines, all before line 363 discards all pending input. Although in principle fixable by a reset option on the write call, Multics currently provides no way to cover this timing window. Fortunately, the window is small enough that most interactive users will go literally for years without encountering an example of a timing failure on input read reset.

429-431 The input and output editing buffer areas are interchanged by these three statements. Here is an example of localizing the use of pointer variables to make clear that they are being used as escapes to allow interchange of the meaning of PL/I identifiers.

440 To go along with the entry point ios_$read_ptr which used stream name user_input by default, Multics does not have a corresponding reset entry with a default stream name. As a result, we must embed the stream name "user_input" in this program.

446 Calls to com_err_ and ioa_ take more setup than most, because each requires passing of argument descriptors so that the subroutine at the other end can figure out how many and what type of arguments have been passed. Since this editor always uses the same arguments to call com_err_, a single call in an internal subroutine avoids having multiple copies of the argument setup code.

457 This editor considers typed-in tab characters to be just as suitable for token delimiters as are blanks. Ideally, tab characters would never reach the editor,

instead having been replaced by blanks by the typewriter input routines. Such complete canonicalization of the input stream would eliminate lines 457-464, but would also require a more sophisticated strategy elsewhere to handle editing of text typed in columns.

477 The cv_dec_ library routine is used here rather than a PL/I language feature, because cv_dec_ will always return a value, even if the number to be converted is ill-formed (in which case it returns zero.) Thus the editor retains complete control over the error comments and messages which will be presented to the user. Such control is essential if one is to construct a well-engineered interface which uses consistent and relevant error messages.

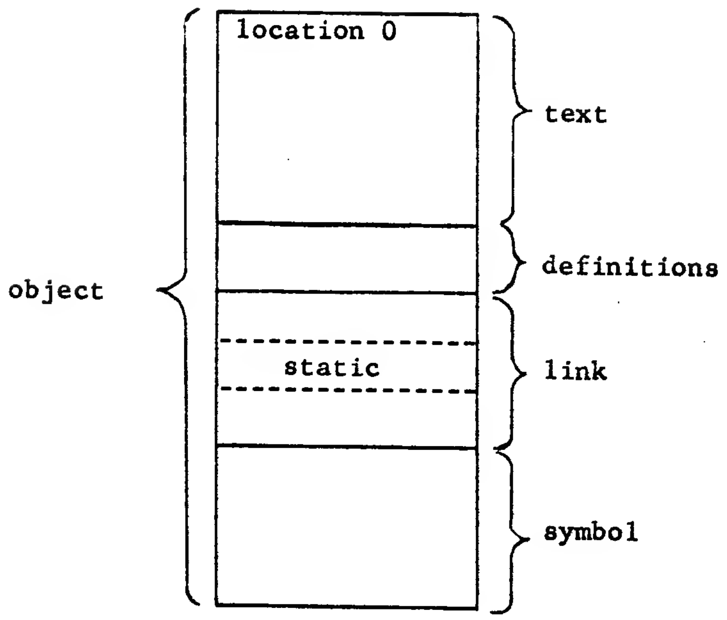
The items printed after the program listing by the compiler do not have line numbers. They are referred to in the following comments by name.

The listing of all variables includes a cross-reference listing, by line number, to facilitate locating all uses of a given variable. This cross-reference listing is also useful for discovering unnecessary variables, which are set and never referenced, or perhaps never referenced at all. Any variable which is referenced only once is suspect, except for external subroutines which may happen to be called only once. Variables never referenced at all appear in the immediately following list. Note that structure names used only as qualifiers (e.g., a.b.c) do not count as uses of the outer names (e.g., a and b). Passing an entire structure as an argument, or structure substitution, would count as a use.

(See listing of identifier alt_lth). The default precision for fixed binary numbers is 17 bits with no fractional part.

"THERE WERE NO NAMES DECLARED BY CONTEXT OR IMPLICATION". This comment was the result of the consistent practice of explicitly declaring everything. If some identifier had not been declared, it would appear in a separate list here, and the compiler would also print a special warning message to the user.

"STORAGE REQUIREMENTS FOR THIS PROGRAM". The result of compiling the above program is the creation of two segments: the listing segment (printed here) and a segment containing a binary machine language program, known as the object segment. The object segment actually contains several different parts, in a format which is interpreted by the mechanisms used for linking to and executing procedures. The numbers printed under this heading require the following picture of an object segment for interpretation:



- *object* is the entire segment.

- *text* is the binary machine language program,

- *definitions* is a set of character string names of entry points to this segment and procedures which it calls.

- *link* is a prototype linkage section, to be copied into the linkage/static segment when this procedure is first used.

- *static* is the part of the prototype linkage section in which PL/I internal static variables are allocated. Initial values for such variables are stored here.

- *Symbol* contains relocation bits for the text and linkage areas, in case this segment is to be permanently bound together with some other object segment. It also contains other things such as the date and time of compilation and, if the table option is specified to the compiler, a symbol table, for debugging. The example shown here did not use the table option, so the symbol section is quite small.

All of the numbers describing storage requirements are printed in octal, so, for example, the binary machine instructions occupy 3015 (octal) locations or 1549 (decimal) locations. Since the program contains about 315 executable statements, each source program line has apparently expanded to an average of about five machine language instructions. The program is shown as using two words of static storage, despite the lack of variables declared to be internal static. The two words of static storage are allocated by the compiler for use by program trace and debugging packages.

Following the object segment description are details about automatic storage allocation. All internal procedures except get_token share automatic storage with the main editor program, which means that fast subroutine calls are compiled to them. Subroutine get_token could have used a fast subroutine call, but the compiler, noting the call to get_token from another internal subroutine (on line 475) conservatively chose to use a full call, since a back call from get_token might have caused recursion. Future versions of the compiler may attempt to trace the flow of such cross calls to guarantee lack of recursion, and thus permit fast calls in more cases.

"THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM." Many frequently used PL/I features are implemented in a library segment named pl1_operators_, and are used by fast subroutine calls compiled into the program. It is useful to get a feeling for what kinds of linguistic constructs result in such calls, by examining a detailed machine language listing some time.

The list of numbers at the end of the program provides a complete map of the machine instructions generated by each statement. This map is useful when debugging following the unexpected printing of a message such as "Out of bounds fault at location 1104 of segment eds."

Although it was not printed here, it is also possible (by using the -list option) to have the compiler print out the detailed machine language program which it generated. Such a printout is useful for reviewing the performance of a program, since it may provide clues about use of PL/I constructs which are inherently expensive to implement.

(text continues on page 55)

```
COMPILATION LISTING OF SEGMENT eds
Compiled by: Multics PL/I Compiler, Version 2 of 15 August 1972.
Compiled on: 09/07/72  2155.0 edt Thu
    Options: optimize map

 1 eds:      procedure;
 2
 3
 4 /*        internal variable declarations.  */
 5
 6 declare   alt_lth    fixed binary;                                 /*  Holds position of next tab.  */
 7 declare   break      character(1) aligned;                         /*  Holds break char for change.  */
 8 declare   brkl       fixed binary;                                 /*  Holds index of change break char.  */
 9 declare   buffer     character(210) aligned;                       /*  Typewriter input buffer.  */
10 declare   code       fixed binary(35);                             /*  For returned status codes.  */
11 declare   count      fixed binary;
12 declare   csize      fixed binary initial(0);
13 declare   edct       fixed binary;
14 declare   ednm       character(3) aligned initial("eds");          /*  Name of the editor, for comments.  */
15 declare   ename      character(32) aligned;                        /*  Holds name of segment being edited.  */
16 declare   exptr      pointer;                                      /*  Temporary pointer holder.  */
17 declare   from_ptr   pointer;                                      /*  Pointer to current from_seg.  */
18 declare   from_seg   character(131072) aligned based(from_ptr);    /*  Editing is from this segment.  */
19 declare   token_lth  fixed binary;                                 /*  Length of token.  */
20 declare   globsw     bit(1) aligned;                               /*  On if "g" option used in change.  */
21 declare   i          fixed binary;
22 declare   ij         fixed binary;
23 declare   indf       fixed binary initial(0);
24 declare   indt       fixed binary initial(0);
25 declare   ioname     character(10) initial("user_input");          /*  Stream name for resetread.  */
26 declare   j          fixed binary;
27 declare   k          fixed binary;
28 declare   l          fixed binary;
29 declare   line       character(210) aligned;                       /*  Holds line currently being edited.  */
30 declare   lngth      fixed binary initial(0);
31 declare   located    fixed binary;
32 declare   m          fixed binary;
33 declare   n          fixed binary;
34 declare   nl         character(1) aligned initial("
35 ");                                                                 /*  Literal "new line" character.  */
36 declare   out_count  fixed binary(24);                              /*  Holds segment bit length.  */
37 declare   out_ptr    pointer;                                       /*  Pointer to out_seg.  */
38 declare   out_seg    character(131072) aligned based(out_ptr);      /*  Outside segment for read or write.  */
39 declare   prc        fixed binary initial(210);                     /*  Size of all buffers.  */
40 declare   sname_lth  fixed binary;                                  /*  Length of source segment name.  */
41 declare   sname_ptr  pointer;                                       /*  Pointer to source segment name.  */
42 declare   status     bit(72) aligned;                               /*  To hold I/O status.  */
43 declare   templ      bit(1) aligned;
44 declare   tlin       character(210) aligned;                        /*  Buffer to hold output of change.  */
45 declare   tkn        character(8) aligned;                          /*  Holds next item on typed line.  */
46 declare   to_seg     character(131072) aligned based(to_ptr);       /*  Editing is to this segment.  */
47 declare   to_ptr     pointer;                                       /*  Pointer to to_seg.  */
48
```

```
/*        external subroutine declarations.  */

declare   com_err_            entry options(variable);
declare   cu_$arg_ptr         entry(fixed binary, pointer, fixed binary, fixed binary(35));
declare   cv_dec_             entry(character(*) aligned)   returns(fixed binary);
declare   expand_path_        entry(pointer, fixed binary, pointer, pointer, fixed binary(35));
declare   hcs_$status_mins    entry(pointer,fixed binary, fixed binary(24), fixed binary(35));
declare   hcs_$make_seg       entry(character(*), character(*), character(*), fixed binary(5),
                                        pointer, fixed binary(35));
declare   hcs_$set_bc_seg     entry(pointer, fixed binary(24), fixed binary(35));
declare   hcs_$truncate_seg   entry(pointer, fixed binary, fixed binary(35));
declare   ioa_                entry options(variable);
declare   ios_$read_ptr       entry(pointer, fixed binary, fixed binary);
declare   ios_$resetread      entry(character(*), bit(72) aligned);
declare   ios_$write_ptr      entry(pointer, fixed binary, fixed binary);

declare   (addr, divide, index, min, null, substr)          builtin;

declare   1 mid based(from_ptr), 2 space(32768) fixed binary, 2 seg(32767) fixed binary;
/*
```

```
*/
/*        .        .        .            P R O G R A M       .        .        .          */

/*  Set up Buffer segments.  */

          ename = "eds_temp";
          call hcs_$make_seg("", "eds_temp", "", 01011b, from_ptr, code);
          if from_ptr = null
               then do;
                    call call_com;
                    return;
                    end;
          to_ptr = addr(mid.seg);

/* Now check to see if an input argument was given */

          call cu_$arg_ptr(1,sname_ptr,sname_lth,code);
          if code ¬=0 then do;
ferror:             ename = "";
                    call call_com;
                    go to quit1;
                    end;

/* Now get a pointer to the segment to be edited */

          call expand_path_(sname_ptr,sname_lth,addr(buffer),addr(ename),code);
          if code ¬= 0 then go to ferror;
          call hcs_$make_seg((buffer),(ename),"",01011b,out_ptr,code);
          if out_ptr = null then go to ferror;
          call hcs_$status_mins(out_ptr,l,out_count,code);
          if code ¬= 0 then go to ferror;
          if out_count = 0 then do;
                    call ioa_("Segment ^a not found.", ename);
                    go to pinput;
                    end;
          csize = divide(out_count,9,17,0);      /* change bit count to char count */
          substr(from_seg,1,csize) = substr(out_seg,1,csize);         /*  Move source segment into buffer.  */
/*  Main editing loop . . . . .  */


pedit:     call ioa_("Edit.");
next:
          call ios_$read_ptr(addr(buffer),prc,count);
          if count=1 then go to next;            /* if null line then get another line, don't print error */
          edct = 1;                     /*  Set up counter to scan this line.  */
          call get_token;                                   /*  Identify next token.  */

          if   tkn = "i"                then go to insert;
          if   tkn = "r"                then go to retype;
          if   tkn = "l"                then go to locate;
          if   tkn = "p"                then go to print;
          if   tkn = "n"                then go to nexlin;
          if   tkn = "save"             then go to file;
          if   tkn = "c"                then go to change;
          if   tkn = "d"                then go to dellin;
          if   tkn = "w"                then go to wsave;
          if   tkn = "t"                then go to top;
          if   tkn = "b"                then go to bottom;
          if   tkn = "."                then go to pinput;

/* If none of the above then not a request */

          call ioa_("'¬a' Not an edit Request", substr(buffer,1,count-1));
          call resetread;
          go to next;

/* ********* input mode ********* */

pinput:   call ioa_("Input.");                   /* print word input */
input:    call ios_$read_ptr(addr(buffer),prc,count);                  /* read a line */
          if substr(buffer,1,1) = "."
               then if count = 2 then go to pedit;                     /* check for mode change */
          call put;
          substr(line,1,count) = substr(buffer,1,count);         /* move line inputted into intermediate storage */
          lngth = count;
          go to input;                                           /* repeat 'til "." */


/* ********* delete ********* */

dellin:
          call get_num;
           do i = 1 to n-1;                                      /* do for each line to be deleted */
               call get;                                         /*  Get next line, overwrite current one.  */
           end;
          lngth = 0;                                             /* nullify last line */
           go to next;

/* ********* insert ********* */

insert:   call put;                                              /*  Add current line to output segment.  */
retype:                                    /*  This is also the retype request.  */
          substr(line,1,count-edct) = substr(buffer,edct+1,count-edct);          /* add replaced line */
           lngth = count - edct;
           go to next;

/* ********* next ********* */

nexlin:   call get_num;
          if n < 0 then go to backup;
          m,j = indf;                                  /* save where you are */
          call put;
          do i = 1 to n;                               /* once for each nl */
            if j>=csize then go to n_eof;              /* check for eof */
            k = index(substr(from_seg,j+1,csize-j),nl);     /*locate end of line */
            if k=0 then do;                            /* no nl (eof) print eof */
n_eof:         if indf>=csize then go to eof;
```

```
               lngth = 0;                                            /* set to no line */
               substr(to_seg,indt+1,csize-m) = substr(from_seg,m+1,csize-m);/*move in top of file */
               indf = csize;
               indt = indt + csize - m;                   /* set pointers */
               go to eof;
               end;
            j = j + k;                                               /* increment j by length of line */
            end;
          indf = j;                                                  /* set pointers and move in top of file */
          lngth = k;
          substr(line,1,k) = substr(from_seg,j-k+1,k);               /* put working line in line */
          substr(to_seg,indt+1,indf-lngth-m) = substr(from_seg,m+1,indf-lngth-m); /* fill rest of file */
          indt = indt + indf - lngth - m;
          go to next;

/* ********* locate ********* */

locate:
          if count=edct then go to incmplt;                          /* check for plain "l NL" */
          edct = edct + 1;                                           /*  Skip delimiter.  */
          j = indt;                                                  /* initialize pointers for index type search */
          m = indf;
          n = csize-indf;
          call put;                                                  /*  Save current line.  */
          if (csize=0) | (n<=0)
              then do;
                    call switch;
                    if j>0 then n = j - 1; else n = 0;
                    m, j = 0;
                    end;
          i = index(substr(from_seg,indf+1,n),substr(buffer,edct,count-edct)); /*locate*/
          if i^=0 then do;                                           /* if found then do */
            do k = indf+i to 1 by -1;                     /* find beginning of line */
               if substr(from_seg,k,1)=nl then go to l_nl;
               end;
            k = 0;
l_nl:
            do indf = k+1 to csize-1 by 1 while(substr(from_seg,indf,1)^=nl);/* find end of line */
               end;
            substr(to_seg,indt+1,k-m) = substr(from_seg,m+1,k-m);/* move in top of file */
            lngth = indf - k;
            indt = indt + k - m;
            substr(line,1,lngth) = substr(from_seg,k+1,lngth);       /* put found line in line */
            n = 1;
            go to printl;                                            /* print found line if wanted */
            end;
          call copy;
          call switch;
          go to eof;
```

```
/* ********* print ********* */

print:
          call get_num;
          if lngth = 0 then                                           /* print indication of no lines */
           do;
               call ioa_("No line.");
               go to noline;
           end;
print1:   call ios_$write_ptr(addr(line),0,min(prc,lngth)); /* write the line */
noline:   n = n-1;
           if n = 0 then go to next;                                  /* any more to be printed? */
          call put;
          call get;
           go to print1;

/* ********* change ********* */

change:
          located = 0;
          if count = 2 then do;
incmplt:    call ioa_("Improper:  ^a", substr(buffer,1,count-1));
          call resetread;
            go to next;
            end;
          brk1 = edct + 2;
          break = substr(buffer,edct+1,1);                 /*  Pick up the delimiting character.  */
          i = index(substr(buffer,brk1,count-brk1),break);
          if i=0 then go to incmplt;
          j = index(substr(buffer,i+brk1,count-brk1-i),break);
          if j=0 then j = count-i-brk1+1;
          edct = edct + i + j + 1;                         /*  Continue scanning edit line.  */
          globsw = "0"b;                         /*  Assume only one change.  */
          n = 1;                                 /*  Assume only one line changed.  */
nxarg:    call get_token;
          if   tkn ^= " " then do;               /*  If token there, process it.  */
              if   tkn = "g" then globsw = "1"b;  /*  Change all occurrances.  */
                    else call cv_num;
              go to nxarg;                       /*  Try for another argument.  */
              end;
          if lngth = 0 then go to skipch;        /*  Skip changing empty line.  */

ch1:      templ = "0"b;                                              /*to indicate if anything was c'd on line */
          m, ij, l = 1;                                              /* indexes to strings */
          if i=1 then do;                                            /* add to begining of line */
            templ = "1"b;
            located = 1;
            substr(tlin,1,j-1) = substr(buffer,brk1+i,j-1); /* copy part to be added */
            substr(tlin,j,lngth) = substr(line,1,lngth);             /* copy old line */
            ij = j + lngth - 1;
            go to cprt;
            end;
ch2:      k = index(substr(line,m,lngth-m),substr(buffer,brk1,i-1)); /*locate what is to be changed */
          if k^=0 then do;
```

```
            substr(tlin,ij,k-1) = substr(line,m,k-1);                  /* copy line up to change */
            substr(tlin,ij+k-1,j-1) = substr(buffer,brkl+i,j-1);/* put in change */
            m = m + k + i - 2;                                         /* increment indexes */
            ij = ij + k + j - 2;
            templ = "1"b;                                              /* indicate that you did someting */
            located = 1;
            if globsw then go to ch2;
            end;
          substr(tlin,ij,lngth-m+1) = substr(line,m,lngth-m+1);        /* copy rest of line */
          ij = ij + lngth - m;
cprt:     if templ then call ios_$write_ptr(addr(tlin),0,ij); /* write if something changed */
          substr(line,1,ij) = substr(tlin,1,ij);
          lngth = ij;
skipch:   if n<=1 then do;                                             /* finished */
            if located=0 then do;
                              call ioa_("Nothing changed by:  ¬a", substr(buffer,1,count-1));/* if not located */
                              call resetread;
                              end;
            go to next;
            end;
          n = n-1;
          call put;
          call get;
          go to ch1;

/* ********* top ********* */

top:      call copy;
          call switch;
           go to next;

/* ********* bottom ********* */

bottom:   call copy;
           lngth = 0;
           go to pinput;

/* ********* backup ********* */

backup:   i = indt;                                                    /* save ptrs */
          call copy;
          call switch;
          indf = i+1;                                                            /* restore ptrs */
          do n = n to 0;                          /*  Note that "n" starts negative.  */
               do indf = indf-1 to 1 by -1;                                      /* look for begining of lines */
                 if substr(from_seg,indf,1) = nl then go to newln;
                 end;
               if n ¬= 0 then do;
               lngth = 0;                                                   /* went off top of file */
               n = 1;
               indt, indf = 0;
               go to eof;
               end;
newln:         end;
```

```
          indt = indf;                                    /* line starts as indt */
          substr(to_seg,1,indt) = substr(from_seg,1,indt);/* move in top of file */
          do indf = indt+1 by 1 to csize;                      /* find end of line */
          substr(line,indf-indt,1) = substr(from_seg,indf,1);  /* move into line */
            if substr(from_seg,indf,1)=nl then go to line_end; /* search for end of line */
            end;
          indf = csize;
line_end: lngth = indf - indt;
          go to next;

/* ********** "file" request ********** */

file:     call copy;                       /*  Finish copy.  */
          call save;
quitl:    call hcs_$truncate_seg(from_ptr,0,code);
          ename = "";
          if code ¬= 0 then call call_com;
          return;

/* ********** write save ********** */

wsave:
          call copy;                       /*  Finish copy.  */
          call save;
          go to next;                      /*  Continue accepting requests.  */

/* ********* eof ********* */

eof:      call ioa_("End of File reached by:¬/¬a", substr(buffer,1,count-1));
          call resetread;
           go to next;

/* ********** FILE SYSTEM ERROR ********** */

error:
          call call_com;
          call resetread;
          go to next;




/*
```

```
*/
/* ********* I N T E R N A L   P R O C E D U R E S ********* */




copy:     procedure;                                                /* copy rest of file into to file */
          substr(to_seg,indt+1,lngth) = substr(line,1,lngth);      /*  Copy current line.  */
          indt = indt + lngth;
          lngth = 0;
          if csize=0 then return;                                   /*  If new input, then no copy needed.  */
          ij = csize - indf;                                        /* do rest of file */
          if ij>0 then substr(to_seg,indt+1,ij) = substr(from_seg,indf+1,ij);
          indt = indt + ij;                                         /* set counters */
          indf = csize;
          return;

          end copy;


save:     procedure;                            /*  Procedure to write out "to" buffer.  */
          call hcs_$truncate_seg(out_ptr,0,code);
          if code ¬= 0 then go to error;
          substr(out_seg,1,indt) = substr(to_seg,1,indt);
          call hcs_$set_bc_seg(out_ptr,indt*9,code);
          if code ¬= 0 then go to error;
          return;

          end save;


put:      procedure;
          substr(to_seg,indt+1,lngth) = substr(line,1,lngth);      /* do move */
          indt = indt + lngth;                                      /* set counters */
          lngth = 0;                                                /*  Discard old line.  */
          return;

          end put;


get:      procedure;                                        /*  Get next line in from_seg into "line".  */
          lngth = 0;                                        /*  Reset current line length.  */
          if indf >= csize then go to eof;                  /*  If no input left, give up.  */
          lngth = index(substr(from_seg,indf+1,csize-indf), nl);     /*  Find the next new line.  */
          if lngth = 0 then lngth = csize-indf;             /*  If no nl found, treat end of segment as one.  */
          substr(line,1,lngth) = substr(from_seg,indf+1,lngth);      /*  Return the line to caller.  */
          indf = lngth+indf;                                /*  Move the "from" pointer ahead one line.  */
          return;

          end get;


switch:   procedure;                                                /* make from-file to file, and v.v. */
               exptr = from_ptr;
```

```
                    from_ptr = to_ptr;
                    to_ptr = exptr;
                    csize = indt;
                    indt,indf = 0;
                    lngth = 0;
                    return;

               end switch;

resetread:               procedure;                         /*  Call I/o system reset read entry.  */
                         call ios_$resetread(ioname,status);   /*  In one place to minimize call setup code.  */
                         return;

                         end resetread;

call_com: procedure;                                        /*  Call com_err_ from standard place.  */
          call com_err_(code,ednm,ename);                   /*  In one place to minimize call setup code.  */
          return;

          end call_com;


get_token:               procedure;
               tkn = " ";                                  /*  Clear out old token.  */
               do edct = edct by 1 to count while (substr(buffer,edct,1) = " ");
                    end;
               token_lth = index(substr(buffer,edct,count-edct)," "); /*  Scan to next blank.  */
               alt_lth = index(substr(buffer,edct,count-edct),"      ");      /*  Look for tab also.  */
               if token_lth+alt_lth = 0                    /*  Pass token back.  */
                    then token_lth = count - edct;                    /*  Neither found, use rest of line.  */
                    else do;                               /*  One or both delimiters were found.  */
                         if token_lth*alt_lth = 0                     /*  Check for both found.  */
                              then token_lth = token_lth+alt_lth-1;   /*  Only one, set alt_lth to it.  */
                              else token_lth = min(token_lth,alt_lth) - 1;      /*  Both found, use smallest.  */
                         end;
                    token_lth = min(8,token_lth);
                    substr(tkn,1,token_lth) = substr(buffer,edct,token_lth);
                    edct = edct + token_lth;
               if alt_lth > 0 then if alt_lth<token_lth then edct = edct - 1;   /*  If initial tah, hack up scanner.  */
               return;

          end get_token;


get_num:  procedure;                                  /*  Routine to convert token to binary integer.  */
          call get_token;                                          /*  Delimit the token.  */
cv_num:   entry;                                                   /*  Enter here if token already available.  */
          n = cv_dec_(tkn);
          if n = 0 then n = 1;                                     /*  Default count is 1.  */
          return;

          end get_num;

          end eds;
```

```
         NAMES DECLARED IN THIS COMPILATION.

IDENTIFIER               OFFSET    LOC STORAGE CLASS   DATA TYPE              ATTRIBUTES AND REFERENCES

NAMES DECLARED BY DECLARE STATEMENT.
addr
                                                       builtin function       internal dcl 65 ref 80 93 93 93 93 110 110 138 138
                                                                              235 235 290 290
alt_lth                            000100 automatic    fixed bin(17,0)        dcl 6 set ref 457 458 461 461 463 468 468
break                              000101 automatic    char(1)                dcl 7 set ref 252 253 255
brkl                               000102 automatic    fixed bin(17,0)        dcl 8 set ref 251 253 253 255 255 256 273 278 281
buffer                             000103 automatic    char(210)              dcl 9 set ref 93 93 95 110 110 131 131 138 138 139
                                                                              142 160 206 247 247 252 253 255 273 278 281 295
                                                                              295 302 362 454 456 457 466
code                               000170 automatic    fixed bin(35,0)        dcl 10 set ref 74 84 85 93 94 95 97 98 348 350 397
                                                                              398 400 401 446
com_err_                           000012 constant     entry                  external dcl 51 ref 446
count                              000171 automatic    fixed bin(17,0)        dcl 11 set ref 110 112 131 131 138 139 142 142 143
                                                                              160 160 162 193 206 246 247 247 253 255 256 295
                                                                              295 362 362 454 456 457 458
csize                              000172 automatic    fixed bin(17,0)        initial dcl 12 set ref 12 103 104 104 172 173 175
                                                                              177 177 178 179 198 200 212 336 340 12 386 387 390
                                                                              418 419 420 432
cu_$arg_ptr                        000014 constant     entry                  external dcl 52 ref 84
cv_dec_                            000016 constant     entry                  external dcl 53 ref 477
divide                                                 builtin function       internal dcl 65 ref 103
edct                               000173 automatic    fixed bin(17,0)        dcl 13 set ref 113 160 160 160 162 193 195 195 206
                                                                              206 251 252 257 257 454 454 454 456 456 457 457
                                                                              458 466 467 467 468 468
ednm                               000174 automatic    char(3)                initial dcl 14 set ref 14 14 446
ename                              000175 automatic    char(32)               dcl 15 set ref 73 86 93 93 95 100 349 446
expand_path_                       000020 constant     entry                  external dcl 54 ref 93
exptr                              000206 automatic    pointer                dcl 16 set ref 429 431
from_ptr                           000210 automatic    pointer                dcl 17 set ref 74 75 80 104 173 177 186 187 206
                                                                              209 212 215 218 325 335 337 338 348 388 419 421
                                                                              429 430
from_seg                                  based        char(131072)           dcl 18 set ref 104 173 177 186 187 206 209 212 215
                                                                              218 325 335 337 338 388 419 421
globsw                             000213 automatic    bit(1)                 dcl 20 set ref 258 262 286
hcs_$make_seg                      000024 constant     entry                  external dcl 56 ref 74 95
hcs_$set_bc_seg                    000026 constant     entry                  external dcl 58 ref 400
hcs_$status_mins                   000022 constant     entry                  external dcl 55 ref 97
hcs_$truncate_seg                  000030 constant     entry                  external dcl 59 ref 348 397
i                                  000214 automatic    fixed bin(17,0)        dcl 21 set ref 97 151 171 206 207 208 253 254 255
                                                                              255 256 257 270 273 278 281 282 319 322
ij                                 000215 automatic    fixed bin(17,0)        dcl 22 set ref 269 275 280 281 283 283 288 289 289
                                                                              290 291 291 292 387 388 388 388 389
index                                                  builtin function       internal dcl 65 ref 173 206 253 255 278 419 456
                                                                              457
indf                               000216 automatic    fixed bin(17,0)        initial dcl 23 set ref 23 169 175 178 184 187 187
                                                                              188 197 198 206 208 212 212 216 322 324 324 325
                                                                              330 334 336 337 337 338 340 341 23 387 388 390 418
                                                                              419 419 420 421 422 422 433
indt                               000217 automatic    fixed bin(17,0)        initial dcl 24 set ref 24 177 179 179 187 188 188
                                                                              196 215 217 217 319 330 334 335 335 336 337 341
```

```
                                                                24 383 384 384 388 389 389 399 399 400 408 409 409
                                                                432 433
ioa_                    000032 constant     entry               external dcl 60 ref 100 109 131 137 232 247 295
                                                                362
ioname                  000220 automatic    char(10)            initial unaligned dcl 25 set ref 25 25 440
ios_$read_ptr           000034 constant     entry               external dcl 61 ref 110 138
ios_$resetread          000036 constant     entry               external dcl 62 ref 440
ios_$write_ptr          000040 constant     entry               external dcl 63 ref 235 290
j                       000223 automatic    fixed bin(17,0)     dcl 26 set ref 169 172 173 173 182 182 184 186 196
                                                                203 203 204 255 256 256 257 273 273 274 275 281
                                                                281 283
k                       000224 automatic    fixed bin(17,0)     dcl 27 set ref 173 174 182 185 186 186 186 208 209
                                                                211 212 215 215 216 217 218 278 279 280 280 281
                                                                282 283
l                       000225 automatic    fixed bin(17,0)     dcl 28 set ref·269
line                    000226 automatic    char(210)           dcl 29 set ref 142 160 186 218 235 235 274 278 280
                                                                288 291 337 383 408 421
lngth                   000313 automatic    fixed bin(17,0)     initial dcl 30 set ref 30 143 154 162 176 185 187
                                                                187 188 216 218 218 230 235 235 266 274 274 275
                                                                278 288 288 289 292 314 328 341 30 383 383 384 385
                                                                408 408 409 410 417 419 420 420 421 421 422 434
located                 000314 automatic    fixed bin(17,0)     dcl 31 set ref 244 272 285 294
m                       000315 automatic    fixed bin(17,0)     dcl 32 set ref 169 177 177 177 179 187 187 187 188
                                                                197 204 215 215 215 217 269 278 278 280 282 282
                                                                288 288 288 289
min                                         builtin function    internal dcl 65 ref 235 235 463 465
n                       000316 automatic    fixed bin(17,0)     dcl 33 set ref 151 168 171 198 200 203 203 206 219
                                                                236 236 237 259 293 300 300 323 323 327 329 477
                                                                478 478
nl                      000317 automatic    char(1)             initial dcl 34 set ref 34 173 209 212 325 338 34
                                                                419
null                                        builtin function    internal dcl 65 ref 75 96
out_count               000320 automatic    fixed bin(24,0)     dcl 36 set ref 97 99 103
out_ptr                 000322 automatic    pointer             dcl 37 set ref 95 96 97 104 397 399 400
out_seg                        based        char(131072)        dcl 38 set ref 104 399
prc                     000324 automatic    fixed bin(17,0)     initial dcl 39 set ref 39 110 138 235 235 39
seg             100000         based        fixed bin(17,0)     array level 2 dcl 67 set ref 80
sname_lth               000325 automatic    fixed bin(17,0)     dcl 40 set ref 84 93
sname_ptr               000326 automatic    pointer             dcl 41 set ref 84 93
status                  000330 automatic    bit(72)             dcl 42 set ref 440
substr                                      builtin function    internal dcl 65 set ref 104 104 131 131 139 142
                                                                142 160 160 173 177 177 186 186 187 187 206 206
                                                                209 212 215 215 218 218 247 247 252 253 255 273
                                                                273 274 274 278 278 280 280 281 281 288 288 291
                                                                291 295 295 325 335 335 337 337 338 362 362 383
                                                                383 388 388 399 399 408 408 419 421 421 454 456
                                                                457 466 466
templ                   000332 automatic    bit(1)              dcl 43 set ref 268 271 284 290
tkn                     000420 automatic    char(8)             dcl 45 set ref 116 117 118 119 120 121 122 123 124
                                                                125 126 127 261 262 453 466 477
tlin                    000333 automatic    char(210)           dcl 44 set ref 273 274 280 281 288 290 290 291
to_ptr                  000422 automatic    pointer             dcl 47 set ref 80 177 187 215 335 383 388 399 408
                                                                430 431
to_seg                         based        char(131072)        dcl 46 set ref 177 187 215 335 383 388 399 408
```

```
token_lth                     000212 automatic      fixed bin(17,0)          dcl 19 set ref 456 458 458 461 461 461 463 463 465
                                                                             465 466 466 467 468

NAMES DECLARED BY DECLARE STATEMENT AND NEVER REFERENCED.
mid                                  based          structure                level 1 unaligned dcl 67
space                                based          fixed bin(17,0)          array level 2 dcl 67

NAMES DECLARED BY EXPLICIT CONTEXT.
backup                        002113 constant       label                    dcl 319 ref 168 319
bottom                        002110 constant       label                    dcl 313 ref 126 313
call_com                      002577 constant       entry                    internal dcl 445 ref 77 87 350 368 445
ch1                           001542 constant       label                    dcl 268 ref 268 303
ch2                           001616 constant       label                    dcl 278 ref 278 286
change                        001362 constant       label                    dcl 244 ref 122 244
copy                          002334 constant       entry                    internal dcl 382 ref 222 307 313 320 346 355 382
cprt                          002007 constant       label                    dcl 290 ref 276 290
cv_num                        002771 constant       entry                    internal dcl 476 ref 263 476
dellin                        000645 constant       label                    dcl 149 ref 123 149
eds                           000102 constant       entry                    external dcl 1 ref 1
eof                           002275 constant       label                    dcl 362 ref 175 180 224 331 362 418
error                         002331 constant       label                    dcl 368 ref 368 398 401
ferror                        000226 constant       label                    dcl 86 ref 86 94 96 98
file                          002245 constant       label                    dcl 346 ref 121 346
get                           002501 constant       entry                    internal dcl 416 ref 152 239 302 416
get_num                       002763 constant       entry                    internal dcl 474 ref 149 167 228 474
get_token                     002626 constant       entry                    internal dcl 452 ref 114 260 452 475
incmplt                       001366 constant       label                    dcl 247 ref 193 247 254
input                         000610 constant       label                    dcl 138 ref 138 144
insert                        000664 constant       label                    dcl 159 ref 116 159
l_nl                          001205 constant       label                    dcl 212 ref 209 212
line_end                      002241 constant       label                    dcl 341 ref 338 341
locate                        001072 constant       label                    dcl 193 ref 118 193
n_eof                         000751 constant       label                    dcl 175 ref 172 175
newln                         002162 constant       label                    dcl 333 ref 325 333
nexlin                        000707 constant       label                    dcl 167 ref 120 167
next                          000417 constant       label                    dcl 110 ref 110 112 133 155 163 189 237 249 298
                                                                             309 342 358 364 371
noline                        001353 constant       label                    dcl 236 ref 233 236
nxarg                         001515 constant       label                    dcl 260 ref 260 264
pedit                         000404 constant       label                    dcl 109 ref 109 139
pinput                        000575 constant       label                    dcl 137 ref 101 127 137 315
print                         001311 constant       label                    dcl 228 ref 119 228
print1                        001330 constant       label                    dcl 235 ref 220 235 240
put                           002460 constant       entry                    internal dcl 407 ref 141 159 170 199 238 301 407
quit1                         002247 constant       label                    dcl 348 ref 88 348
resetread                     002560 constant       entry                    internal dcl 439 ref 132 248 296 363 370 439
retype                        000665 constant       label                    dcl 160 ref 117 160
save                          002413 constant       entry                    internal dcl 396 ref 347 357 396
skipch                        002036 constant       label                    dcl 293 ref 266 293
switch                        002543 constant       entry                    internal dcl 428 ref 202 223 308 321 428
top                           002105 constant       label                    dcl 307 ref 125 307
wsave                         002272 constant       label                    dcl 355 ref 124 355

THERE WERE NO NAMES DECLARED BY CONTEXT OR IMPLICATION.
```

```
STORAGE REQUIREMENTS FOR THIS PROGRAM.

          Object     Text      Link      Symbol    Defs      Static
Start          0        0      3162        3224    3015        3172
Length      3456     3015        42         217     145           2

External procedure eds uses 496 words of automatic storage
Internal procedure copy shares stack frame of parent block
Internal procedure save shares stack frame of parent block
Internal procedure put shares stack frame of parent block
Internal procedure get shares stack frame of parent block
Internal procedure switch shares stack frame of parent block
Internal procedure resetread shares stack frame of parent block
Internal procedure call_com shares stack frame of parent block
Internal procedure get_token uses 68 words of automatic storage
Internal procedure get_num shares stack frame of parent block

THE FOLLOWING EXTERNAL OPERATORS ARE USED BY THIS PROGRAM.
r_e_as              r_le_a              alloc_cs            move_csa            csa_move            call_ext_out_desc
call_ext_out        call_int_this       return              set_csa             set_cs_co           cs_move_co
shorten_stack       blank_csa           index_cs_co         index_cs_1_co       ext_entry           int_entry
rpd_loop_1_lp_bp    rpd_loop_1_bp_lp

THE FOLLOWING EXTERNAL ENTRIES ARE CALLED BY THIS PROGRAM.
com_err_                      cu_$arg_ptr                   cv_dec_                       expand_path_
hcs_$make_seg                 hcs_$set_bc_seg               hcs_$status_mins              hcs_$truncate_seg
ioa_                          ios_$read_ptr                 ios_$resetread                ios_$write_ptr

NO EXTERNAL VARIABLES ARE USED BY THIS PROGRAM.


  LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC       LINE    LOC
     1 000100         12 000107         14 000110         23 000112         24 000113         25 000114         30 000121
    34 000122         39 000124         73 000126         74 000134         75 000174         77 000200         78 000201
    80 000202         84 000205         85 000224         86 000226         87 000231         88 000232         93 000233
    94 000256         95 000260         96 000327         97 000333         98 000350         99 000352        100 000354
   101 000375        103 000376        104 000400        109 000404        110 000417        112 000434        113 000437
   114 000441        116 000445        117 000452        118 000457        119 000464        120 000471        121 000476
   122 000503        123 000510        124 000515        125 000522        126 000527        127 000534        131 000541
   132 000572        133 000574        137 000575        138 000610        139 000625        141 000634        142 000635
   143 000642        144 000644        149 000645        151 000646        152 000657        153 000660        154 000662
   155 000663        159 000664        160 000665        162 000703        163 000706        167 000707        168 000710
   169 000712        170 000715        171 000716        172 000726        173 000731        174 000747        175 000751
   176 000754        177 000755        178 001003        179 001005        180 001011        182 001012        183 001013
   184 001015        185 001017        186 001021        187 001035        188 001064        189 001071        193 001072
   195 001075        196 001076        197 001100        198 001102        199 001105        200 001106        202 001115
   203 001116        203 001124        204 001125        206 001127        207 001157        208 001161        209 001167
   210 001201        211 001204        212 001205        214 001231        215 001233        216 001261        217 001264
   218 001270        219 001303        220 001305        222 001306        223 001307        224 001310        228 001311
   230 001312        232 001314        233 001327        235 001330        236 001353        237 001355        238 001357
   239 001360        240 001361        244 001362        246 001363        247 001366        248 001417        249 001421
   251 001422        252 001425        253 001435        254 001454        255 001456        256 001476        257 001505
```

```
258 001512    259 001513    260 001515    261 001521    262 001526    263 001536    264 001537
266 001540    268 001542    269 001543    270 001547    271 001552    272 001554    273 001556
274 001576    275 001611    276 001615    278 001616    279 001653    280 001656    281 001677
282 001732    283 001737    284 001744    285 001746    286 001750    288 001752    289 002003
290 002007    291 002027    292 002034    293 002036    294 002042    295 002044    296 002075
298 002077    300 002100    301 002102    302 002103    303 002104    307 002105    308 002106
309 002107    313 002110    314 002111    315 002112    319 002113    320 002115    321 002116
322 002117    323 002122    324 002127    325 002135    326 002147    327 002152    328 002154
329 002155    330 002157    331 002161    333 002162    334 002164    335 002166    336 002172
337 002203    338 002227    339 002235    340 002237    341 002241    342 002244    346 002245
347 002246    348 002247    349 002263    350 002266    351 002271    355 002272    357 002273
358 002274    362 002275    363 002326    364 002330    368 002331    370 002332    371 002333
382 002334    383 002335    384 002351    385 002353    386 002354    387 002357    388 002361
389 002406    390 002410    391 002412    396 002413    397 002414    398 002430    399 002432
400 002437    401 002455    402 002457    407 002460    408 002461    409 002475    410 002477
411 002500    416 002501    417 002502    418 002503    419 002506    420 002524    421 002533
422 002540    423 002542    428 002543    429 002544    430 002546    431 002550    432 002552
433 002554    434 002556    435 002557    439 002560    440 002561    441 002576    445 002577
446 002600    447 002621    452 002622    453 002633    454 002636    455 002661    456 002664
457 002703    458 002710    461 002721    463 002731    465 002737    466 002744    467 002751
468 002753    469 002762    474 002763    475 002764    476 002770    477 002772    478 003007
479 003013
```

Handling Large Files on Multics

A frequent point of confusion about Multics concerns the handling of large data files within the segmented virtual memory environment. A file, in Multics terminology is a (usually structured) collection of data of arbitrary size. A file which happens to require less than 256K words of storage is usually stored in a single segment of the Multics storage system, and is addressed by mapping the segment containing the entire file into the current address space. Source and object programs, and small, linear ASCII text files are examples of files handled this way. A file which is larger than 256K words (or which is smaller but may someday grow that large) is usually stored in several segments in a single directory in the Multics storage system, and is addressed by mapping relevant parts (records) of the file into the current address space. The directory contains, in addition to the raw data of the file, any maps or indexes needed to maintain its internal organization. Three file management facilities (sometimes called Access Methods on IBM systems) are available to handle the details of setting up, indexing, and searching of files. These are:

1. Multi-segment files (MSF): There is a system-wide standard format for ASCII text files which require more than 256K words of storage. Most translators, for example, are prepared to produce very long output listings for the printer using this format; the high speed line printer facilities also recognize the format.

2. File manager: A general purpose, record-oriented file manipulation system provides sequential record files and indexed (keyed) record files of up to 100 million bytes. The files are accessed using the virtual memory: one calls to the file manager giving the index or key of the record desired; the file manager returns a pointer to the location of that record in the address space, and the program then can manipulate the contents of the record using, for example, a PL/I based structure. The file manager provides interlocking facilities for multiple users, and also guarantees integrity of a file in the case where a system failure occurs while the user is updating the file. The MPM reference guide section on the file manager, and write-ups of a set of subroutines beginning with the name fm_ should be consulted for further information.

3. PL/I record-oriented I/O: The full ANSI standard PL/I I/O system is implemented on Multics*, allowing construction of a data manipulation system which is in principle system independent. Since the PL/I I/O

system uses the Multics File Manager (2, above) very large files can be efficiently set up, updated, and searched using only the PL/I language. For further information, one should consult the PL/I language specifications.

In addition, users with unusually sophisticated needs such as completely inverted files, files with indexes on different elements, etc., will find that appropriate facilities can easily be developed using the virtual memory combined with techniques similar to those used by the Multics File Manager. It is important to realize that the Multics File Manager, while organized as a protected subsystem, is written in PL/I, using only Multics facilities which are also available to the user. Thus, a user could construct his own version of the File Manager, or a more elaborate file accessing system without recourse to special privileges or need to modify the Multics supervisor.

Finally, the Multics I/O system, which is organized to allow attachment of arbitrary source-sink I/O devices, may be used to read and write magnetic tape in any of several formats, for applications in which permanent on-line storage is not appropriate.

Unfortunately, there does not yet exist a suitable set of annotated case studies on the use of the file management facilities. The potential developer of a large file application is advised to begin by reviewing one or more applications previously implemented on Multics and which use these tools.

CS-TR Scanning Project
Document Control Form

Date : 2/29/96

Report # LCS-TR-123

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:

Document Information

Number of pages: 218(223-IMAGES
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:

☐ Typewriter	☐ Offset Press	☐ Laser Print
☐ InkJet Printer	☒ Unknown	☐ Other:

Check each if included with document:

☒ DOD Form	☐ Funding Agent Form	☐ Cover Page
☐ Spine	☐ Printers Notes	☐ Photo negatives
☐ Other:		

Page Data:

Blank Pages(by page number):

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:

IMAGE MAP: (1-218) UN#'ED TITLEPAGE, ii-vii, UN#BLANK, ix-x, 1-1 THRU 1-16, 2-1 THRU 2-39, UN#BLK, 2-41-2-71, UN#BLK, 2-73 THRU 2-101, UN#BLK, 3-1 THRU 3-33, UN#BLK, 4-1 THRU 4-56
(219-223) SCANCONTROL, DOD, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 2/29/96 Date Scanned: 2/29/96 Date Returned: 1/6/96

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd

BIBLIOGRAPHIC DATA SHEET	1. Report No. MAC TR-123	2.	3. Recipient's Accession No.
4. Title and Subtitle Introduction to Multics			5. Report Date: Issued February 1974
			6.
7. Author(s) Jerome H. Saltzer			8. Performing Organization Rept. No. MAC TR-123
9. Performing Organization Name and Address PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY: 545 Technology Square, Cambridge, Massachusetts 02139			10. Project/Task/Work Unit No.
			11. Contract/Grant No. N00014-70-A-0362-0006
12. Sponsoring Organization Name and Address Office of Naval Research Department of the Navy Information Systems Program Arlington, Va 22217			13. Type of Report & Period Covered: Interim Scientific Report
			14.

15. Supplementary Notes: This report is a snapshot of the Introduction to the Users' Manual for the Multics system. The complete users' manual is available from the M.I.T. Information Processing Center.

16. Abstracts

This report is an introduction to the properties, concepts, and usage of the Multics system. Its four chapters are designed for reading continuity rather than for reference or completeness. Chapter 1 provides a broad overview. Chapter 2 goes into the concepts underlying Multics. Chapter 3 is a tutorial guide to the mechanics of using the system, with illustrative examples of terminal sessions. Chapter 4 provides a series of examples of programming in the Multics environment.

17. Key Words and Document Analysis. 17a. Descriptors

Time-Sharing Systems
Computer Utilities
Operating Systems
Multics
Multiple-Access Computers

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 213
	20. Security Class (This Page) UNCLASSIFIED	22. Price